45

A Computational Logic for Applicative Common LISP

MATT KAUFMANN AND J. STROTHER MOORE

1 Introduction

Perhaps one of the most ambitious goals for mathematical logic was put forth by one of its earliest advocates.

> If we had some exact language . . . or at least a kind of truly philosophic writing, in which the ideas were reduced to a kind of alphabet of human thought, then all that follows rationally from what is given could be found by a kind of calculus, just as arithmetical or geometrical problems are solved. (Leibniz, 1646–1716)

Mathematical logic casts too harsh a light to be appropriate for the 'rationalization' of many human endeavors. Can one axiomatize good and evil, or even the aerodynamics of the African sparrow, so that all that follows by mathematical proof is truly believable?

But Leibniz' dream was aided immeasurably by the invention of the digital computer because the computer not only provided a platform on which to build a reasoning engine but provided a source of problems to tackle with it.

> Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. (John McCarthy, "A Basis for a Mathematical Theory of Computation," 1961)

Computing systems, such as microprocessors, switches, file servers, compilers, encryption devices, control programs, financial software, etc., are naturally described in the precise language of mathematical logic. If the logical 'model' of the system accurately describes what is built, then the logical properties of the model accurately predict the behavior of the artifact.

But is proving theorems about computing systems practical? Is it cost effective? Here are two more quotations that shed some light on those questions.

> An elusive circuitry error is causing a chip used in millions of computers to generate inaccurate results. (*NY Times*, "Circuit Flaw Causes Pentium Chip to Miscalculate, Intel Admits," November 11, 1994)

> Intel Corp. last week took a $475 million write-off to cover costs associated with the divide bug in the Pentium microprocessor's floating-point unit. (*EE Times*, January 23, 1995)

It is possible to prove a lot of theorems for $475 million.

'ACL2' stands for 'A Computational Logic for Applicative Common Lisp.' It is the name of a programming language, a first-order mathematical logic based on recursive functions, and a mechanical theorem prover for that logic. ACL2 is designed for use in reasoning about computing systems, both those implemented in hardware and those implemented in software.

The human user of ACL2 can formalize or model a computing system by defining functions that simulate the operation of the system. Since ACL2 is a programming language, such an operational model is just a computer program that can be run on concrete data to produce concrete results. With this program the user might test the behavior of the system on some finite number of example inputs. Since ACL2 is also a mathematical logic, the user might prove theorems about the model, possibly establishing properties that hold for an infinite number of inputs. Finally, using ACL2's interactive theorem proving program, the user might check these proofs mechanically, thereby eliminating the all-too-frequent errors that crop up in 'hand proofs.'

This is not just a mathematical fantasy. For example, ACL2 was used to prove the correctness of the circuitry implementing the elementary floating point operations on the AMD Athlon™ processor[1] with ACL2. Most major chip manufacturers have personnel devoted to proving theorems or otherwise formally checking properties of their designs.

In this article we describe ACL2 briefly, present a simple modeling problem and its solution in ACL2, and describe some of ACL2's recent applications.

We assume the reader has had a little experience with computing and programming. Also helpful would be an introductory course in first order predicate calculus.

2 The ACL2 System

Here we briefly discuss ACL2 as a programming language, a logic, and a mechanical theorem prover or proof checker. The ACL2 system is available under the GNU General Public License and without fee from its home page, `http://www.-cs.utexas.edu/users/moore/ac12`. Installation instructions and documentation are included. We discuss how to learn to use ACL2 in Section 5.

ACL2 is just one of several mechanical theorem proving programs used for hardware and software verification. Among the others are HOL (Gordon and Melham 1993), Otter (McCune 1994), and PVS (Owre et al. 1992). See the *Related Web Sites* link under the *Books and Papers* link of the ACL2 home page for lists of dozens of other theorem provers. Theorem provers are still research vehicles, even though some, like the ones mentioned above, are being used by researchers in industry. Each is designed to explore a different part of the theorem proving problem. ACL2 is first order with considerable automation, with heuristics tailored to recursive definitions and induction. Otter supports first-order predicate calculus, with full support for quantification. HOL and PVS both support higher-order logics. Of major concern to the developers of HOL

was how to build a theorem prover that was both user-extensible and sound. The PVS and ACL2 developers were primarily concerned with building tools that people without research backgrounds in automated reasoning could use off the shelf to prove theorems about computing systems. The Otter team focused on finding automatic proof techniques so that Otter, rather than its human users, gets full credit for its proofs. We should emphasize, however, that all of these tools address all of these issues to varying degrees. For example, ACL2 addresses user extensibility, and Otter requires the user to interact by setting parameters.

The programming language

As a programming language, ACL2 is a variant of Common Lisp (Steele 1990). 'Lisp,' which stands for 'list processing,' is commonly used for artificial intelligence applications because it facilitates symbol manipulation. Lisp was invented by John McCarthy in the late 1950s as part of his visionary project towards a mechanized theory of computation (McCarthy 1960, 1962, 1963).

ACL2 is a *functional* or *applicative* version of Lisp, meaning that ACL2 programs are mathematical functions of their arguments. They do not have side-effects and are not sensitive to implicit 'global variables' or implicit 'state.'

ACL2 terms are written in prefix notation. A term is a variable, a constant, or the application of a function symbol, f, of k arguments to k terms, $a_1, \ldots, a_k$, written $(f\ a_1 \ldots a_k)$. Here is how one might write $a^2 + ab$ in ACL2: `(+ (expt a 2) (* a b))`. The ACL2 runtime system provides facilities for calculating the values of terms under assignments of values to their free variables. For example, if `a` has the value `3` and `b` has the value `5`, then the term above is calculated to have the value `24`.

In addition to the numbers (integers, rationals, and complex numbers with rational coefficients) ACL2 supports several other data types. These include strings (such as '`Hello World`'), symbols (such as `LOAD` and `X`), and ordered pairs. Primitive functions are provided for manipulating each type of data. For example, the function `cons` takes two arguments and returns an ordered pair containing them. The functions `car` and `cdr` take one argument, which is normally an ordered pair, and return the first and second components, respectively. The function `consp` takes one argument and returns the constant `T` ('true') if the argument is an ordered pair and `NIL` ('false') otherwise.

Ordered pairs are written in parenthesized 'dot notation.' For example, the pair traditionally written as $\langle$`3 , NIL`$\rangle$ is written in ACL2 as `(3 . NIL)`.

Ordered pairs can be used to encode a wide variety of abstractions. One such abstraction is linear lists, which are so common that the notation for printing ordered pairs in ACL2 (and Lisp) is oriented towards it. The constant `NIL` may be written simply as `()`, and thus plays double duty; it is used both as the false truth-value and as the empty list. The ordered pair `(3 . NIL)` may be written simply as `(3)`. The ordered pair `(2 . (3 . NIL))` may be written `(2 3)`, the ordered pair `(1 . (2 . (3 . NIL)))` may be written `(1 2 3)`, etc.

It is convenient to be able to write list constants inside terms. What is a term that evaluates to (i.e. whose meaning is) the list `(1 2 3)`? One such term is `(cons 1 (cons 2 (cons 3 NIL)))`. But another one is `'(1 2 3)`. The 'quote mark' can be used to write a term that evaluates to a given constant.

Lists are frequently used to represent still other abstractions. For example, the list `(1 2 3)` may be thought of as the *stack* obtained by pushing `1` onto the stack `(2 3)`. The following *function definitions* make these conventions easier to remember. (Actually, the symbols `push` and `pop` are defined in ACL2 and may not be redefined; to make these definitions we must actually operate in a symbol *package* other than the default one, but we do not discuss that here.)

```
(defun push    (item stack)    cons item stack))
(defun top     (stack)         (car stack))
(defun pop     (stack)         (cdr stack))
```

The first `defun` above, for example, defines `push` to be a function symbol of two arguments, `item` and `stack`, whose value is obtained by evaluating the term `(cons item stack)`. Thus, `(push 1, '(2 3))` is the stack `(1 2 3)`. The `top` of that stack is `1` and the `pop` is `(2 3)`.

Here is another common use of lists. Consider the list of two ordered pairs `((A . 7) (B . 4))`. Call this constant α. The `car` of α is `(A . 7)`. The `cdr` of α is `((B . 4))`. The `car` of the `cdr` of α is `(B . 4)`. Lists such as α are thought of as *tables* that map keys (`A` and `B`, in this case) to values (`7` and `4`, respectively). Such lists are called *association lists* or *alists* or *assignments*. The `car` of the `car` of a nonempty alist is the first key assigned in the alist; the `cdr` of the `car` is the value assigned to that key. The `cdr` of a nonempty alist is another alist that assigns the rest of the symbols.

Here is a function that looks up the value of a symbol in an alist. This function is recursive.

```
(defun lookup (sym alist)
(if (consp alist)
(if (equal sym (car (car alist)))
    (cdr (car alist))
    (lookup sym (cdr alist)))
0))
```

The definition above may be paraphrased as: If `alist` is a `cons` pair, then if `sym` is the first key assigned, return its value; otherwise `lookup sym` in the rest of `alist`. If `alist` is not a `cons` pair, return `0`.

After this definition, `(lookup 'B 'α)` evaluates to `4`. But `(lookup 'C 'α)` evaluates to `0`.

ACL2 supports a variety of syntactic extensions. Another way to define lookup is shown below.

```
(defun lookup (sym alist)
  (cond
    ((endp alist) 0)
    ((equal sym (car (car alist)))
     (cdr (car alist)))
    (t (lookup sym (cdr alist)))))
```

The cond special form is just a nest of ifs. `(Endp alist)` is equivalent to `(not (consp alist))`.

For a more thorough introduction to ACL2 as a programming language see Kaufman et al. (2000b). The link labeled *Hyper-Card* on the ACL2 home page contains a quick introduction to Lisp and a reference card to the programming language. The *User's Manual* link contains several megabytes of hypertext documentation.

The Logic

ACL2 is formalized as a first-order mathematical logic. Any standard formulation of first-order logic will serve our purposes. See also Kaufmann and Moore (to appear). Axioms describe the primitive functions. For example, here are several of the axioms. (Actually, `(consp NIL) = NIL` is not an axiom but an easily proved theorem.)

Axioms

```
x = NIL → (if x y z) = z.
x ≠ NIL → (if x y z) = y.
(consp NIL) = NIL
(consp (cons x y)) = t
(car (cons x y)) = x
(cdr (cons x y)) = y
```

Using the natural numbers and ordered pairs, a representation of the ordinals up to ε_0 is introduced. For example, the ordinal $\omega^2 + \omega \times 4 + 3$ is represented in ACL2 by the list `(2 1 1 1 1 . 3)`. An axiom defines a relation (a function returning `T` or `NIL`), named `e0-ord-<`, corresponding to the well-founded ordering relation on these ordinals. Another axiom introduces the predicate, `e0-ordinalp`, which recognizes the ACL2 ordinals.

The principle of mathematical induction, in ACL2, is then stated as a rule of inference that allows induction up to ε_0. To prove a conjecture by induction one must identify some ordinal-valued measure function. The induction principle permits one to assume inductive instances of the conjecture being proved, provided the instance has a smaller measure according to the chosen measure function.

Finally, a principle of definition is provided, by which the user can extend the axioms by the addition of equations defining new function symbols. To admit a new recursive definition, the principle requires the identification of an ordinal measure function and a proof that the arguments to every recursive call decrease according to this measure. Only terminating recursive definitions can be so admitted under the definitional principle. ('Partial functions' can be axiomatized; see Maniolos and Moore (2000).)

The successful admission of a definition adds a new axiom. For example, the definition of `push` above adds

Axiom

```
(push item stack) = (cons item stack).
```

The two variables are (implicitly) universally quantified.

The measure used to justify the recursive function `lookup` is `ac12-count`. The `ac12-count` of a natural number is that number. The `ac12-count` of an ordered pair is one more than sum of the counts of the `car` and the `cdr`. `Ac12-count` always returns a natural number.

As is customary in formal treatments of mathematical logic, from such basics a variety of other rules of inference are derived to make proofs more practical. A more thorough treatment of the logic is presented in Kaufmann et al. (2000b: Chapter 6). The solutions to the exercises for that chapter (see the *Books and Papers* on the home page and follow the obvious links) contain formal proofs of many elementary theorems and a sketch of how more elaborate rules of inference are justified. See also Kaufmann and Moore (1997).

The theorem prover

The ACL2 theorem prover is a symbolic manipulation engine driven from a collection of rules in a database. The user determines the available rules, but in an indirect way. The rules are derived from theorems posed as challenges by the user and proved by the system. Thus, the logical soundness of the theorem prover cannot be imperiled by the user. But the strategy employed by the theorem prover can be largely determined by the experienced user who understands how rules are derived from theorems and what the effects of those rules are. The user may also direct the system to read in all the rules in previously certified 'books,' thereby enabling the sharing of results in the ACL2 community. In addition, the user may supply hints to affect the system's decisions and may specify low-level proof steps via an interactive loop.

The system has many heuristics for determining its behavior. For example, heuristics determine when it expands recursive function definitions, when it inducts, and what induction hypotheses it assumes.

The system also contains many decision procedures and other high-level derived rules of inference. For example, it can use a BDD procedure (Bryant 1992) to recognize propositional tautologies, it has built in knowledge about linear arithmetic inequalities (Boyer and Moore 1997), and it can use calculation to compute the values of functions on constants.

The system prints a description of its evolving 'proof' as it proceeds. It does not produce a formal proof, but when it says 'Q.E.D.' we, the authors of ACL2, believe that the computation it did is sufficient to guarantee the existence of a formal proof in the logic described. The system often fails, either by abandoning the proof attempt or running until the user aborts the attempt. In either case, it is up to the human user to 'fix' the situation, by reformulating the conjecture to prove or the hints provided, or by further developing a database of rules.

The ACL2 theorem prover is an improved version of the Boyer–Moore theorem prover, Nqthm (Boyer and Moore 1979, 1997; Boyer et al. 1995), adapted to applicative Common Lisp. For more details of how it works, see Kaufmann et al. (2000b). We illustrate it in the section entitled "Sample Output" below.

How good is ACL2's theorem prover? That is, how automatic is it? At one level that depends on how good a database of rules it has and whether the conjecture at hand

falls in the class of formulas handled by that set of rules. But perhaps the intent of the question is deeper. How far away from its rules can it operate successfully? How deep are its proofs? The answer depends on whether you view the question from the perspective of the logician or the more traditional mathematician, who have very different ideas of what the word 'proof' means. Logicians think of proofs as sequences or trees of formulas, expressed in a precisely defined syntax and related to one another by a precisely defined set of inference rules. Most mathematicians think of proofs as informal but convincing arguments. The logician might very well consider ACL2 an automatic theorem prover because it is not always obvious how to construct formal proofs of some of the theorems it proves automatically. But the mathematician would probably think of ACL2 as a proof checker, at best. The mathematician would find virtually everything ACL2 proves automatically to be 'self-evident' or 'obvious' from the theorems and definitions ACL2 had previously been led to accept. To the mathematician, ACL2 is not so much *finding* a proof as it is *checking* one presented to it by the human. This will become more obvious in Section 3.

3 A Modeling Problem

In this section we will deal with a simple variant of a classic example in the verification literature, first done 'by hand' in McCarthy and Painter (1967) and by machine with the Boyer–Moore prover in Boyer and Moore (1979). We will model an assembly language for a push-down stack machine, formalize a simple arithmetic expression language, implement a compiler that translates from arithmetic expressions to assembly code, and prove the compiler is correct. In addition to illustrating the formalization of some central ideas in computing – state machines, language semantics, compilation – this example is appropriate because it deals with a few of the same issues that arise in model theory, for example the assignment of meaning to the sentences of a formal language.

The assembly language

An *instruction* is a nonempty list. The *opcode* is the first element of the list. Some instructions have an *operand*, which is the second element.

```
(defun opcode (inst) (car inst))
(defun operand (inst) (car (cdr inst)))
```

The opcodes on our machine and their informal semantics are: `(LOAD` *var*`)` pushes the value of *var* onto the stack, `(PUSH` *c*`)` pushes the constant *c* onto the stack, `(DUP)` duplicates the top of the stack, `(ADD)` pops two items off the stack and pushes their sum, and `(MUL)` pops two items off the stack and pushes their product. We formalize this with the function `step`, which takes an instruction to execute, an alist giving the variable values, and a stack; the function returns the new value of the stack.

```
(defun step (inst alist stk)
  (let ((op (opcode inst)))
    (cond
   ((equal op 'LOAD)
    (push (lookup (operand inst) alist) stk))
   ((equal op 'PUSH)
    (push (operand inst) stk))
   ((equal op 'DUP)
    (push (top stk) stk))
   ((equal op 'ADD)
    (push (+ (top (pop stk)) (top stk))
             (pop (pop stk))))
   ((equal op 'MUL)
    (push (* (top (pop stk)) (top stk))
             (pop (pop stk))))
(t stk))))
```

A *program* is a sequence of instructions. They are executed sequentially with a given alist and some initial stack. The final stack is returned.

```
(defun m (program alist stk)
  (cond ((endp program) stk)
        (t (m (cdr program)
              alist
              (step (car program) alist stk)))))
```

The function `m` formalizes the semantics of this simple programming language. For example,

```
(m '((LOAD A) (DUP) (ADD))
   '((A . 7) (B . 4))
   '(1 2 3))
```

'simulates' the execution of a program that pushes the value of `A`, duplicates it, and adds the two values together. It does so in an environment in which the value of `A` is 7 and the value of `B` is 4. The initial stack is `(1 2 3)`, a stack with 1 on top. The result of this execution is the stack `(14 1 2 3)`.

An expression language

An *expression* (and its *value* under an assignment) is a variable symbol (whose value is specified by the assignment), a numeric constant (which is its own value), or a list of one of the following forms (where the $expr_i$ are expressions): `(INC` $expr_1$`)` (whose value is one more than that of $expr_1$), `(SQ` $expr_1$`)`, (whose value is the square of that of $expr_1$), `(`$expr_1$ + $expr_2$`)` (whose value is the sum of those of the two

subexpressions), or $(expr_1 * expr_2)$ (whose value is the product of those of the two subexpressions).

We can formalize this as follows.

```
(defun eval (x alist)
  (cond
  ((atom x)
   (cond ((symbolp x) (lookup x alist))
         (t x)))
  ((equal (fn x) 'INC)
   (+ 1 (eval (argl x) alist)))
  ((equal (fn x) 'SQ)
   (* (eval (argl x) alist)
      (eval (argl x) alist)))
  ((equal (fn x) '+)
   (+ (eval (argl x) alist)
      (eval (arg2 x) alist)))
  ((equal (fn x) '*)
   (* (eval (arg1 x) alist)
      (eval (arg2 x) alist)))
  (t 0)))
```

where

```
(defun fn (expr)
  (if (equal (len expr) 2) (car expr) (car (cdr expr))))
(defun arg1 (expr)
  (if (equal (len expr) 2) (car (cdr expr)) (car expr)))
(defun arg2 (expr)
  (car (cdr (cdr expr)))).
```

Eval formalizes the semantics of this expression language. We can test it. For example, here is a transcript showing that the `eval` of a certain expression is equal to `400`.

```
COMP ! > (eval '(SQ (INC (A + (3 * B)))) '((A. 7) (B. 4)))
400
COMP ! >
```

A compiler

A compiler is a translator from one language to another. We will compile arithmetic expressions, as above, into our assembly language. The goal is to produce a program that, when executed under a given assignment, will push the value of the expression on the stack. The method is straightforward. To compile a product, say, we concatenate

the compiled code for the two subexpressions and then generate an (MUL) instruction to pop the two intermediate values off the stack and push their product. To compile (SQ $expr_1$), we will compile the subexpression and then generate a (DUP) followed by an (MUL). The others are similar. Here is the compiler.

```
(defun compile (x)
  (cond
   ((atom x)
    (cond
     ((symbolp x) (list (list 'LOAD x)))
     (t (list (list 'PUSH x)))))
   ((equal (fn x) 'INC)
    (append (compile (arg1 x))
            '((PUSH 1) (ADD))))
   ((equal (fn x) 'SQ)
    (append (compile (arg1 x))
            '((DUP) (MUL))))
   ((equal (fn x) '+)
    (append (compile (arg1 x))
            (compile (arg2 x))
            '((ADD))))
   ((equal (fn x) '*)
    (append (compile (arg1 x))
            (compile (arg2 x))
            '((MUL))))
   (t (list (list 'PUSH 0)))))
```

Append concatenates its arguments. We illustrate the compiler below.

Specification

The output of compile on the expression (SQ (INC (A + (3 * B)))) is the program shown below.

```
COMP ! > (compile '(SQ (INC (A + (3 * B)))))
((LOAD A)
 (PUSH 3)
 (LOAD B)
 (MUL)
 (ADD)
 (PUSH 1)
 (ADD)
 (DUP)
 (MUL))
COMP ! >
```

This program is 'correct' in the sense that executing it leaves the value of the given expression on top of the stack.

The *specification* of compile is that it produces correct programs for every expression. A formalization of this claim is `(equal top (m (compile x) a s)) (eval x a))`. We will name this conjecture `main`.

Mechanical proof

All of the definitions involved in the formalization above are automatically admitted by the mechanical theorem prover. `Ac12-count` is the only measure needed and the system 'guesses' that.

If we then submit `main` as a challenge conjecture, the ACL2 theorem prover runs for 11 seconds (on a 731 MHz Pentium III) and gives up. Inspection of the proof attempt using 'The Method,' described in Kaufmann et al. (2000b) and in the on-line ACL2 manual, produces the following insights. First, the proof will clearly involve induction on the form of the expression `x`. Second, `main` is not strong enough to prove by induction. We must prove the conjecture that says 'execution of the compiled code *pushes* the value of the expression onto the pre-existing stack (leaving the other items there intact).' Our `main` does not insure that other intermediate values are not removed and hence cannot be used to explain how the compiler works. Note that it is a common mathematical trick to generalize a conjecture before doing proof by induction, and although ACL2 provides a little support for making such generalizations automatically, it is generally up to the user to do so.

The stronger conjecture is `(equal (m (compile x) a s) (push (eval x a) s))`, which, if proved, clearly implies `main`. We name this conjecture `lemma`. The attempt to prove `lemma` fails in about 6 seconds. Inspection of the failed proof reveals that ACL2 chose an inadequate induction scheme. Consider the case for compiling a sum expression. The theorem prover inductively assumes `lemma` for both subexpressions. But it is obvious to the human user that the second induction hypothesis (that for the second argument of the sum) must use the instance in which the stack `s` is the pre-existing with one more thing pushed onto it: the value of the first subexpression.

Induction schemes are described to ACL2 by defining recursive functions that instantiate their arguments appropriately. Here is the necessary definition, which is admitted automatically.

```
(defun hintfn (x a s)
  (cond
   ((atom x) (list x a s))
   ((equal (fn x) 'INC)
    (hintfn (arg1 x) a s))
   ((equal (fn x) 'SQ)
    (hintfn (arg1 x) a s))
   ((equal (fn x) '+)
    (cons (hintfn (arg1 x) a s)
          (hintfn (arg2 x) a (push (eval (arg1 x) a) s))))
   ((equal (fn x) '*)
```

```
    (cons (hintfn (arg1 x) a s)
          (hintfn (arg2 x) a (push (eval (arg1 x) a) s))))
   (t (list x a s))))
```

The value of this function is irrelevant. What matters is the case analysis it does and the way it instantiates its arguments in recursion.

If we then tell ACL2 to prove `lemma`, advising it to induct the way `hintfn` recurs, the proof attempt again fails. Inspection reveals that the system must be able to simplify `(m (append x y) a s)`. This is obvious in retrospect: the compiler concatenates two recursively obtained code sequences and we must know how the machine deals with concatenated programs. The obvious relationship is given in the theorem below.

```
(defthm composition
  (equal (m (append x y) a s)
         (m y a (m x a s))))
```

This is how the user actually submits a challenge to the theorem prover. The formula above *alleges* that the execution of the concatenation of program `x` followed by program `y` is equal to the execution of program `y` starting with the stack produced by the execution of program `x`. If ACL2 can prove this, it will build it in as a rewrite rule (by default) and name the theorem `composition`. In fact, the system successfully proves `composition`, by induction on `x` and simplification. We show the output under "Sample output" below.

The system can now prove `lemma`, and can then use it to prove `main`. The two commands, in full, are shown below.

```
(defthm lemma)
  (equal (m (compile x) a s)
         (push (eval x a) s))
  :hints (("Goal" :induct (hintfn x a s))))
(defthm main
  (equal (top (m (compile x) a s))
         (eval x a)))
```

The total amount of time to replay the entire successful proof sequence (including the admission of all of the definitions) is approximately 2 seconds. All the necessary user input has been exhibited here. An experienced ACL2 user might well have recognized the importance of `composition` and `lemma` from the outset and would thus have stated them as part of the initial proof plan. We mention the discovery process because it is important in more complicated proofs where all the necessary lemmas are rarely recognized in advance.

Sample output

Here is the output of the theorem prover on the `composition` theorem.

```
COMP ! > (defthm composition
                 (equal (m (append x y) a s)
                        (m y a (m x a s))))
```

Name the formula above *1.
Perhaps we can prove *1 by induction. Three induction schemes are suggested by this conjecture. These merge into two derived induction schemes. However, one of these is flawed and so we are left with one viable candidate.

We will induct according to a scheme suggested by (M X A S), but modified to accommodate (APPEND X Y). If we let (:P A S X Y) denote *1 above then the induction scheme we'll use is

```
(AND (IMPLIES (AND (NOT (ENDP X))
                   (:P A (STEP (CAR X) A S) (CDR X) Y))
              (:P A S X Y))
     (IMPLIES (ENDP X) (:P A S X Y))).
```

This induction is justified by the same argument used to admit M, namely, the measure (ACL2-COUNT X) is decreasing according to the relation EO-ORD-< (which is known to be well-founded on the domain recognized by EO-ORDINALP). Note, however, that the unmeasured variable S is being instantiated. When applied to the goal at hand the above induction scheme produces the following two nontautological subgoals.
Subgoal *1/2

```
(IMPLIES (AND (NOT (ENDP X))
              (EQUAL (M (APPEND (CDR X) Y)
                        A (STEP (CAR X) A S))
                     (M Y A (M (CDR X) A (STEP (CAR X) A S)))))
         (EQUAL (M (APPEND X Y) A S)
                (M Y A (M X A S)))).
```

By the simple :definition ENDP we reduce the conjecture to
Subgoal *1/2'

```
(IMPLIES (AND (CONSP X)
              (EQUAL (M (APPEND (CDR X) Y)
                        A (STEP (CAR X) A S))
                     (M Y A (M (CDR X) A (STEP (CAR X) A S)))))
         (EQUAL (M (APPEND X Y) A S)
                (M Y A (M X A S )))).
```

But simplification reduces this to T, using the :definitions BINARY-APPEND, M, OPCODE, OPERAND, POP, PUSH, STEP and TOP, the :executable-counterpart of EQUAL, primitive type reasoning and the :rewrite rules CAR-CONS, COR-CONS, COMMUTATIVITY-OF-* and COMMUTATIVITY-OF-+.
Subgoal *1/1

```
(IMPLIES (ENDP X)
         (EQUAL (M (APPEND X Y) A S)
                (M Y A (M X A S )))).
```

By the simple :definition ENDP we reduce the conjecture to
Subgoal *1/1'

```
(IMPLIES (NOT (CONSP X))
         (EQUAL (M (APPEND X Y) A S)
                (M Y A (M X A S )))).
```

But simplification reduces this to T, using the :definitions BINARY-APPEND and M and primitive type reasoning.
That completes the proof of *1.
Q. E. D.
Summary

```
Form:  ( DEFTHM COMPOSITION . . .)
Rules: ((:DEFINITION BINARY-APPEND)
        (:DEFINITION ENDP)
        (:DEFINITION M)
        . . . material deleted . . .
        (:REWRITE CDR-CONS)
        (:REWRITE COMMUTATIVITY-OF-*)
        (:REWRITE COMMUTATIVITY-OF-+))
Warnings: None
Time: 0.09 seconds (prove: 0.06, print: 0.02, other: 0.01)
COMPOSITION
```

4 Case Studies

The compiler example illustrates two different models, a theorem relating them, and the role of the user in structuring ACL2's proofs by the discovery of appropriate lemmas. At `http://www.cs.utexas.edu/users/moore/publications/flying-demo/scipt.html` you will find this example and many others, including the correctness of an insertion sort function, the correctness of a binary adder and of a multiplier, the formal semantics of a simple netlist description language – a language like that used to describe circuits – and the correctness of a function that generates a description of an adder, and some theorems abut Java byte code programs. The web pages show the definitions, many example computations, and most of the proofs, including all of the proofs for the compiler example discussed here.

These models are suggestive of how ACL2 is used. But they are trivial by the standards of industrial machine designs and realistic programming languages. The stack machine above is fully specified in about two dozen lines of code; the proof required two lemmas. Industrial applications of ACL2 have involved hundreds of pages of code to formalize a single model and thousands of lemmas to relate two such models.

We now briefly describe a few such applications. For more details, see Kaufmann et al. (2000a) and Kaufmann and Moore (2000), collections of case studies written by ACL2 users.

ACL2 has been used to model several industrial microprocessors. The models are similar to that for `m`: a 'state' is formalized as an *n*-tuple of various components like stacks, registers, etc., and a state-transition function, `step`, is defined. The Motorola CAP digital signal processor (DSP) (Brock et al. 1996; Brock and Hunt 1997; Gilfeather et al. 1994) was modeled at two levels: the pipeline level, where several instructions are simultaneously being decoded and carried out; and the user level, where instructions are executed sequentially. Both models were bit- and cycle-accurate in the sense that they specified all the state components completely on every step. The two models were shown equivalent under certain conditions on the program being executed. Another commercial microprocessor modeled with ACL2 is the Rockwell JEM1 (Greve and Wilding 1998; Wilding et al. to appear) – the world's first silicon Java Virtual Machine. ACL2 has been used to verity commercial DSP (Brock and Moore 1999) microcode. It has been used to prove the IEEE compliance of the `FDIV` microcode for the AMD-K5^{TM} processor[2] (Moore et al. 1998) and of the circuit descriptions implementing each of the elementary floating-point operations on the AMD Athlon (Russinoff 1998; Russinoff and Flatau 2000). It has been used to verify a pipelined machine providing interrupts and exceptions in the face of speculative out-of-order execution (Sawada and Hunt 1998) and a security model for the boot code of the IBM 4758 (Smith and Austel 1998).

Not all of ACL2's applications are at the hardware level. ACl2 is being used to prove properties of Java byte code (Moore 1999; Moore and Porter 2000a, 2000b), including multi-threaded programs.

ACL2 has been used to provide a trusted (verified) proof-checker for the Otter theorem proving system (McCune and Shumsky 2000). Otter is perhaps the preeminent resolution-style theorem prover and has been under development at Argonne National Labs for decades. When Otter claims success, it can give its proof to a much simpler theorem prover for checking, and one such checker was verified to be sound for finite models by ACL2. In a similar kind of work, ACL2 was used to verify a checker for an off-line compiler for safety-critical train-borne real-time control software (Bertoli and Traverso 2000).

ACL2 has been used to prove the correctness of a model checker (Manolios 2000), the alternating-bit protocol (Manolios et al. 1999), a BDD package (Sumners 2000), and many other algorithms.

An extension of the system by Ruben Gamboa (1999) adds the real numbers via nonstandard analysis and many interesting theorems in real analysis have been proved including trigonometric identities, Euler's identity, the fundamental theorem of calculus (Kaufmann 2000) and theorems about continuity and differentiability (Gamboa 2000). See also Gamboa and Kaufmann (1999).

The ACL2 home page contains links to many other papers reporting ACL2 applications.

Notes

1 AMD, the AMD logo, and combinations thereof, and AMD Athlon are trademarks of Advanced Micro Devices, Inc.

2 AMD, the AMD logo, and combinations thereof, and AMD-K5 are trademarks of Advanced Micro Devices, Inc.

References

Bertoli, P. and Traverso, P. (2000) Design verification of a safety-critical embedded verifier. In Kaufmann et al. (2000) pp. 233–46.

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

Boyer, R. S. and Moore, J. S. (1988) Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence 11* (pp. 83–124). Oxford: Oxford University Press.

Boyer, R. S. and Moore, J. S. (1997) *A Computational Logic Handbook*. 2nd edn. New York: Academic Press.

Boyer, R. S., Kaufmann, M. and Moore, J. S. (1995) The Boyer–Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 5(2), 27–62.

Brock, Bishop and Warren Hunt, A. Jr. (1997) Formally specifying and mechanically verifying programs for the Motorola complex arithmetic processor DSP. In *1997 IEEE International Conference on Computer Design* (pp. 31–6). IEEE Computer Society, October.

Brock, B. and Moore, J. S. (1999) A mechanically checked proof of a comparator sort algorithm. Submitted for publication.

Brock, B., Kaufmann, M. and Moore, J. S. (1996) ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri (eds.), *Formal Methods in Computer-Aided Design (FMCAD'96)* (pp. 275–93). New York: Springer-Verlag, LNCS 1166, November.

Bryant, R. E. (1992) Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*.

Gamboa, R. (1999) Mechanically verifying real-valued algorithms in ACL2. PhD thesis, University of Texas at Austin.

Gamboa, R. (2000) Continuity and differentiability. In Kaufmann et al. (2000) pp. 301–17.

Gamboa, R. and Kaufmann, M. (forthcoming) Non-standard analysis in ACL2. *Journal of Automated Reasoning.*

Gilfeather, S., Gehman, J. and Harrison, C. (1994) Architecture of a complex arithmetic processor for communication signal processing. In *International Symposium on Optics, Imaging, and Instrumentation, 2296, Advanced Signal Processing: Algorithms, Architectures, and Implementations V* (pp. 624–5). SPIE.

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

Greve, D. A. and Wilding, M. M. (1998) Stack-based Java a back-to-future step. *Electronic Engineering Times*, Jan. 12, p. 92.

Kaufmann, M. (2000) Modular proof: The fundamental theorem of calculus. In Kaufmann et al. (2000), pp. 75–92.

Kaufmann, Matt and Moore, J. (1997) A precise description of the ACL2 logic. In http://www.cs.utexas.edu/users/moore/publications/km97.a.ps.Z. Department of Computer Sciences, University of Texas at Austin.

Kaufmann, M. and Moore, J. S. (2000) *Proceedings of ACL2 Workshop 2000*. Department of Computer Sciences, Technical Report TR-00-29. http://www.cs.utexas.edu/ftp/pub/techreports/tr00-29.dir.

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

Kaufmann, M., Manolios, P. and Moore, J. S. (eds.) (2000a) *Computer-Aided Reasoning: ACL2 Case Studies*. Dordrecht: Kluwer Academic Press.

Kaufmann, M., Manolios, P. and Moore, J. S. (2000b) *Computer-Aided Reasoning: An Approach*. Dordrecht: Kluwer Academic Press.

Manolios, P. (2000) Mu-calculus model-checking. In Kaufmann et al. (2000) pp. 93–112.

Manolios, P. and Moore, J. Partial functions in ACL2. In Kaufmann and Moore (2000). http://www.cs.utexas.edu/ftp/pub/techreports/tr00-29.dir.

Manolios, P., Namjoshi, K. and Sumners, R. (1999) Linking theorem proving and model-checking with well-founded bisimulation. In *Computed Aided Verification, CAV '99* (pp. 369–79). New York: Springer-Verlag, LNCS 1633.

McCarthy, J. (1960) Recursive functions of symbolic expressions and their computation by machine (part I). *CACM*, 3(4), 184–95.

McCarthy, J. (1962) Towards a mathematical science of computation. In *Proceedings of IFIP Congress* (pp. 21–8). Amsterdam: North-Holland.

McCarthy, J. (1963) A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. Amsterdam: North-Holland.

McCarthy, John and Painter, James (1967) Correctness of a compiler for arithmetic expressions. In *Proceedings of Symposia in Applied Mathematics*, vol. 19. American Mathematical Society.

McCune, W. (1994) *Otter 3.0 Reference Manual and Guide*, Technical Report ANL-94/6, Argonne National Laboratory, Argonne, IL. See also URL http://www.mcs.-anl.gov/AR/otter/.

McCune, W. and Shumsky, O. (2000) Ivy: A preprocessor and proof checker for first-order logic. In Kaufmann et al. (2000), pp. 265–82.

Moore, J. S. (1999) Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen (eds.), *Correct System Design – Recent Insights and Advances* (pp. 139–62). LNCS 1710.

Moore, J. S. and Porter, G. (2000a) An executable formal JVM thread model. Submitted for publication.

Moore, J. S. and Porter, G. (2000b) Mechanized reasoning about Java threads via a JVM thread model. Submitted for publication.

Moore, J. S., Lynch, T. and Kaufmann, M. (1998) A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9), 913–26.

Owre, S., Rushby, J. and Shankar, N. (1992) PVS: A prototype verification system. In D. Kapur (ed.), *11th International Conference on Automated Deduction (CADE)*, (pp. 748–52). Lecture Notes in Artificial Intelligence, vol. 607. New York: Springer-Verlag.

Russinoff, D. (1998) A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1, 148–200.

Russinoff, D. M. and Flatau, A. (2000) RTL verification: A floating-point multiplier. In Kaufmann et al. (2000), pp. 201–32.

Sawada, J. and Hunt, W. (1998) Processor verification with precise exceptions and speculative execution. In *Computed Aided Verification, CAV '98* (pp. 135–46). New York: Springer-Verlag. LNCS 1427.

Smith, S. W. and Austel, V. (1998) Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *The Third USENIX Workshop on Electronic Commerce*, September.

Steele, G. L. Jr. (1990) *Common Lisp: The Language*, 2nd edn. Burlington, MA: Digital Press.
Sumners, R. (2000) Correctness proof of a BDD manager in the context of satisfiability checking. In Kaufmann and Moore (2000). http://www.cs.utexas.edu/ftp/pub/techreports/tr00–29.dir.
Wilding, Matthew, Greve, David and Hardin, David (to appear) Efficient simulation of formal processor models. *Formal Methods in System Design*. Draft TR available as http://pobox.com/users/hokie/docs/efm.ps.

Further Reading

If you are interested in reading more about ACL2, the definitive book is Kaufmann et al. (2000b), which explains the programming language, the logic, the theorem prover, and how to use them. The book contains exercises, and the solutions to the exercises are available on the Web through the ACL2 home page http:/www.cs.utexas.edu/users/moore/acl2. A wealth of additional reading material is available from the home page.

In addition, ACL2 is available at no fee under the GNU General Public License. You may install it and then define functions, execute them, and learn to prove theorems with the ACL2 theorem prover. Installation instructions and several megabytes of hypertext documentation are available on the ACL2 home page. Also of value are the *two short tours* link on the home page and the previously mentioned flying demo, http://www.cs.utexas.edu/users/moore/publications/flying-demo/script.html.