Logical Foundations for the ACL2 Theorem Prover

Matt Kaufmann
The University of Texas at Austin
Dept. of Computer Science

Joint work with Bob Boyer, J Moore, and the ACL2 community

Presented at JAF 2019
It’s a bit odd to be giving a talk about a software system to mathematical logicians.

Once upon a time I was one of you . . . but I’ve gone to the dark side.

Now I work on software, ACL2, that proves theorems.

**QUESTION**: What can I say today that might interest you?

**MY ANSWERS:**

1. Introduce ACL2 as a **practical application** of logic.
2. Discuss **foundational issues** for ACL2.
Overview and Context

Introduction to the ACL2 System

Logical Foundations for ACL2

Conclusion
The ACL2 home page begins with the following summary.

ACL2 is a logic and programming language in which you can model computer systems, together with a tool to help you prove properties of those models. “ACL2” denotes “A Computational Logic for Applicative Common Lisp”.

But before we talk about ACL2, let’s put it in context.
Formal verification (FV) of hardware and software systems is the use of tools to check their correctness using mathematical methods, notably proof.

FV tools include equivalence checkers, model checkers, various static checkers, and (occasionally) interactive theorem provers (ITPs) such as Coq, Isabelle, HOL4, PVS, Agda — and ACL2.
Interactive Theorem Proving

- Yearly ITP conference
- ITP is typically more scalable than fully automatic tools, but it requires human assistance.
  - In ACL2, one proves lemmas that may be used automatically to simplify terms in later proofs.

Some strengths of ACL2 among ITPs:

- Proof automation and debugging
- Fast execution of programs
- Documentation in hypertext format (120,000 lines for system; many more for libraries)
- Scalability (see next slide)
ON ACL2 APPLICATIONS

ACL2 has been used not only at universities and the U.S. Government, but also at several companies [4]:

- AMD, ARM, ArterisIP, Battelle, Centaur, GE, IBM, Intel, NXP, Kestrel, Oracle, Rockwell Collins

People are actually paid to prove theorems with ACL2.

“Microprocessor design goes daily through numerous optimizations that affect thousands of lines of code. These optimizations must be proved correct.”

— Anna Slobodova, verification manager, Centaur Technology

A recent example of an ACL2 formalization at UT Austin: An efficient checker for Boolean satisfiability (SAT) proofs

- Used in recent international SAT competitions
- Has checked 2-petabyte SAT proof of longstanding open problem (Schur number 5) [3]; ~16 CPU years
**Overview and Context**

**Introduction to the ACL2 System**

**Logical Foundations for ACL2**

**Conclusion**

**Partial Timeline**

- 1970: Boyer and Moore meet
- 1975: Insertion sort
- 1980: Binary adder
- 1985: Expression compiler
- 1990: Prime factorization
- 1995: BDX930 abandoned
- 2000: RSA
- 2005: Unsolvability of halting problem
- 2010: AMD K5 floating-point division μcode
- 2015: IBM floating point algorithms

- **Gödel**
- **FM8502**
- **Gauss**
- **Unity**
- **FM9001**
- **FM9801**
- **Paris-Harrington Ramsey**
- **Motorola CAP**
- **DEC alpha**
- **Nqthm compiler**
- **Kalman filters**

- **Y86**
- **x86 ring model/proof**
- **Byzantine Generals**
- **micro Gypsy compiler**
- **Kit OS kernel**
- **Piton**
- **biphase mark**
- **clock sync**
- **AMD floating-point rtl, ongoing**
- **Logic formalization (Spain)**
- **Galois/Rockwell SHADE**
- **ACM Software System Award**
- **Y86 with STOBJ**
- **X86 ISA**
- **Dijkstra shortest path**
- **UCLID integration prototype**
- **AAMP7G MIL cert.**
Overview and Context

Introduction to the ACL2 System

Logical Foundations for ACL2

Conclusion
Overview and Context

Introduction to the ACL2 System

Logical Foundations for ACL2

Conclusion
INTRODUCTION TO THE ACL2 SYSTEM

- ACL2 is freely available with libraries of certifiable books.
  - Available from the ACL2 home page and Github
  - Libraries provide more than 500,000 events (theorems, definitions, other).

- ACL2 is written mostly in itself (!).
  - About 11 MB of source files

- ACL2 community holds workshops: #15 held Nov. 2018

- History of the ACL2 system
  - Boyer-Moore Theorem Provers go back to their collaboration starting in 1971. [10]
  - The ACL2 community contributes with feature requests and (on occasion) prototype implementations.
Let’s get familiar with ACL2 (and its syntax): first demo **programming**, then **theorem proving**.

- ACL2 programming and evaluation
  
  [DEMO]: file demo-1.lsp
  (log demo-1-log.txt)

- ACL2 as an automated theorem prover
  
  [DEMO]: file demo-2.lsp
  (log demo-2-log.txt)

  - ACL2 provides **automation** for induction, linear arithmetic, Boolean reasoning, rule application, ...  
  - During a proof, each goal is replaced by a list of subgoals (possible empty) such that if they are all theorems, then that goal is a theorem.
OUTLINE

Overview and Context

Introduction to the ACL2 System

Logical Foundations for ACL2

Conclusion
Overview and Context

Introduction to the ACL2 System

Logical Foundations for ACL2

Conclusion
The ACL2 logic is a first-order logic with $\varepsilon_0$-induction.

Probably weaker induction would usually suffice in practice; maybe only $\omega^\omega$; maybe only each of $\omega, \omega^\omega, \omega^\omega^\omega$, etc., iterated through only standard natural numbers . . .

- . . . but it hasn’t been a priority to consider this, let alone to consider effects on the implementation.

(Anyhow, it’s nice to have Ken Kunen’s Nqthm proof of the Paris-Harrington theorem. [9])
Logical Foundations (2)

Restriction: ACL2 theories extend the ground-zero theory: essentially PA with $\varepsilon_0$-induction, extended with data types.

- numbers (complex rationals);
- characters;
- strings;
- symbols; and
- closure under an ordered pair operation, \texttt{cons}.

\texttt{Cons} provides lists, with the symbol \texttt{nil} for the empty list.

ACL2 \(\texttt{cons 3 nil}\)
(3)
ACL2 \(\texttt{cons 2 (cons 3 nil)}\)
(2 3)
ACL2 \(\texttt{cons 1 (cons 2 (cons 3 nil))}\)
(1 2 3)
ACL2 \(\texttt{cons 0 nil}\)
Logical Foundations (3)

Theory extensions made with ACL2 are conservative (no new theorems in the existing language).

- ... This holds even for recursive definitions, since “termination” must be provable.
- We will see the importance of introducing new concepts locally: justified by conservativity.
- Theories evolve by introducing new function symbols using the extension principles. [6]
EXTENSION PRINCIPLE: DEFINITIONS

A definition extends the current theory with the axiom equating the call with the body. Example (from first demo):

(defun fact (n) ; factorial
  (if (posp n) ; n is a positive integer
      (* n (fact (- n 1)))
      1))

This adds the following axiom (and of course induction axioms):

(fact n) =
(if (posp n) ; n is a positive integer
      (* n (fact (- n 1)))
      1)

A definition may be recursive if some measure into $\varepsilon_0$ is proved to decrease on each recursive call.
EXTENSION PRINCIPLE: CHOICE (AND $\exists$)

Quantification is implemented using a choice operator. When asked to define

$$P(\vec{x}) = \exists \vec{y} A(\vec{x}, \vec{y})$$

then ACL2 generates the following.

**Conservatively introduce** a Skolem (witness) function $w(\vec{x})$ and a predicate $P(\vec{x})$:

$$w(\vec{x}) = \varepsilon \vec{y} A(\vec{x}, \vec{y}) \quad [\text{If any } \vec{y} \text{ satisfies } A(\vec{x}, \vec{y}), \text{ then } w(\vec{x}) \text{ does.}]$$

$$P(\vec{x}) = A(\vec{x}, w(\vec{x}))$$

(defun-sk fermat-counterex (n)
  (exists (i j k)
    (and (posp i) (posp j) (posp k)
      (equal (+ (expt i n) (expt j n))
        (expt k n))))

(deffthm fermat
  (implies (and (integerp n) (< 2 n))
    (not (fermat-counterex n))))
Extension principle: Choice (and $\exists$) (2)

This sort of thing is clearly conservative (we have countable theories, so we don’t even need Choice)...

... IF we ignore induction!

Conservativity with induction follows from a model-theoretic forcing argument.
EXTENSION PRINCIPLE: CONSTRAINTS

It is also legal to introduce \textit{constrained} functions, using axioms that are \textit{proved} about \textit{local witnesses}.

\textbf{Example:}

\begin{verbatim}
(encapsulate ((fn (x y) t))
  (local (defun fn (x y)
          (+ x y)))
  (defthm fn-commutative
          (equal (fn x y) (fn y x))))
\end{verbatim}

A derived inference rule, \textit{functional instantiation} \cite{2}, is often useful with constrained functions.

\textbf{Example:}
(defun map2-fn (lst1 lst2)
  (if (consp lst1)
      (cons (fn (first lst1) (first lst2))
            (map2-fn (rest lst1) (rest lst2)))
      nil))
(defthm map2-fn-commutative
  (implies (equal (len lst1) (len lst2)) ; same length
           (equal (map2-fn lst2 lst1)
                   (map2-fn lst1 lst2))))

(defun map2-* (lst1 lst2)
  (if (consp lst1)
      (cons (* (first lst1) (first lst2))
            (map2-* (rest lst1) (rest lst2)))
      nil))
(defthm map2-**-commutative
  (implies (equal (len lst1) (len lst2))
           (equal (map2-* lst2 lst1)
                   (map2-* lst1 lst2)))
  :hints ("Goal" :by (:functional-instance
                     map2-fn-commutative
                     (fn *) (map2-fn map2-**))))
Conservativity and local

Fun example in ACL2(r), a variant of ACL2 that supports the real numbers, due to Ruben Gamboa: The Overspill Principle of non-standard analysis.

Informally:
If internal predicate $P(n, x)$ holds for all standard natural numbers $n$, then $P(n, x)$ holds for some non-standard natural number $n$.

- **overspill.lisp**: Relatively concise formalization (which I’ll flash on the next slide)
  25 lines

- **overspill-proof.lisp**: Ugly proof (shows need for human assistance), but local to the main proof, by conservativity
  256 lines

Using local can dramatically speed up book inclusion!
(local ; Hence skipped when including this top-level book!
   (include-book "overspill-proof")

(defstub overspill-p (n x) t)

(defun overspill-p* (n x)
  (if (zp n)
      (overspill-p 0 x)
      (and (overspill-p n x)
           (overspill-p* (1- n) x))))

(defchoose overspill-p-witness (n) (x)
  (or (and (natp n) (standardp n)
           (not (overspill-p n x)))
      (and (natp n) (i-large n)
           (overspill-p* n x))))

(defthm overspill-p-overspill
  (let ((n (overspill-p-witness x)))
    (or (and (natp n) (standardp n)
             (not (overspill-p n x)))
        (and (natp n) (i-large n)
             (implies (and (natp m)
                           (<= m n))
                      (overspill-p m x))))):rule-classes nil)
**Meta-theoretic Reasoning (1)**

In ACL2, you can [1, 5]:

- code a simplifier,
- prove that it is sound, and
- direct its use during later proofs.

**Efficient execution** can be important for meta-theoretic reasoning!

A comment in the ACL2 sources, the “Essay on Correctness of Meta Reasoning”, works out the correctness argument.
**ITERATION**

Useful for programming, with reasoning support. **Examples:**

ACL2 !>(loop$ for i in '(3 5 7) sum (* i i))
83
ACL2 !>

ACL2 gives the following semantics to the second of these.

```
(sum$ '(lambda (i) (* i i))
  '(3 5 7))
```

where `sum$` is defined essentially as follows.

```
(defun sum$ (fn lst)
  (if (endp lst) ; lst is empty
      0
    (+ (apply$ fn (list (first lst)))
      (sum$ fn (rest lst))))
```
"HIGHER-ORDER" Apply$ (1)

We cannot employ the usual two-sorted, weak second-order approach. **Example**: Not a theorem without the `defun`!

```
(local (defun f (x) x))
(thm (equal (apply$ 'f (list x)) x))
```

**Example successful use of** `apply$`:

```
(include-book "projects/apply/top" :dir :system)
(defun$ norm^2 (x y) (+ (* x x) (* y y)))
(assert-event (equal (norm^2 3 4) 25))
(thm (equal (norm^2 3 4) 25))
(assert-event (equal (apply$ 'norm^2 (list 3 4)) 25))
```

**But the following fails, as it should:**

`apply$` is a constrained function with trivial constraints.

```
(thm (equal (apply$ 'norm^2 (list 3 4)) 25))
```
"**Higher-Order**" \texttt{Apply$\ (2)$}

However, the proof succeeds for the \texttt{thm} below, where the \textit{warrant hypothesis}, \texttt{(warrant norm^2)}, \textit{asserts}:
\[
(\forall \ x \ y) \ (\text{equal} \ (\text{apply$\ 'norm^2 (\text{list} \ x \ y))}
\]
\[
(\text{norm^2} \ x \ y)\).
\]

\[
\text{(thm} \ (\text{implies} \ (\text{warrant norm^2})
\]
\[
(\text{equal} \ (\text{apply$\ 'norm^2 (\text{list} \ 3 \ 4))}
\]
\[
25))\)
\]

\textbf{Warrant hypotheses are not vacuous!}
There is a natural \textit{evaluation theory} where every warrant is \textit{attached} to the constant "true" function. [8]
Defattach (1)

Defattach provides a way to evaluate constrained functions by giving them new definitions. But it allows extensions that are not conservative. Example:

- **Constraint for** a “specification” function, spec:
  \[ x \in \mathbb{Z} \implies \text{spec}(x) \in \mathbb{Z} \]

- **Define** function \( f \): \( f(x, y) = \text{spec}(x + y) \)

- **Define** an “implementation” function, impl:
  \( \text{impl}(x) = 10 * x \)

- **Attach** impl to spec: (defattach spec impl)

  **Meaning:** \((\forall x)(\text{spec}(x) = \text{impl}(x))\)

Result not provable from axioms for \( f \) and spec:

```
ACL2  !>(f 3 4); = spec(7) = impl(7)
70
ACL2  !>
```
DEFATTACH (2)

Issues to consider:

► Is (local (defattach ...)) supported? YES, local is supported.

► Then how do we deal with conservativity?
  Two theories: The current theory for reasoning and a stronger evaluation theory, extended using defattach:

  \[(\forall x)(\text{spec}(x) = \text{impl}(x))\]

► Ah, but what about this?

  \[(\text{thm} \ (\text{equal} \ (f \ 3 \ 4) \ 70))\]

  The proof fails! (Good!)

► Is the evaluation theory consistent?
  Yes, where the attachment relation must be acyclic.

Details: see Essay on Defattach comment in the ACL2 sources.
Some More Logical Challenges

Practical considerations create some more logical challenges.

- **Packages** are a programming convenience but introduce axioms such as the following: *not conservative!*
  
  ```lisp
  symbol-package-name('PKG1::F) = "PKG1"
  ```
  Hence packages **must be recorded**.

- One can specify a measure in order to admit a recursive definition. But what if the measure is defined in terms of a function whose definition is **LOCAL**?

- **Congruence-based reasoning** allows replacing one subterm by another that is equivalent but not necessarily equal. [7]
Overview and Context

Introduction to the ACL2 System

Logical Foundations for ACL2

Conclusion
**CONCLUSION**

- ACL2 has a 29 (or 48) year history and is used in industry.
- As an ITP system, it relies on user guidance for large problems but enjoys scalability.
- Logic provides critical foundational support for practical theorem proving software.
- For more information, see the [ACL2 home page](#), in particular links to [The Tours](#) and [Publications](#), which links to introductory material.


Matt Kaufmann
matthew.j.kaufmann@gmail.com

Slides for this talk are available via links from my home page:
http://www.cs.utexas.edu/users/kaufmann

THANK YOU!
Extra Slides

We can go on, time permitting....
Some ACL2 features *not* discussed further today:

- **Prover algorithms**
  - Waterfall, linear arithmetic, Boolean reasoning, …
  - **Rewriting**: Conditional, congruence-based, rewrite cache, syntaxp, bind-free, …

- **Using the prover effectively**
  - The-method and introduction-to-the-theorem-prover
  - Theories, hints, rule-classes, …
  - Accumulated-persistence, brr, proof-checker, dmr, …

- **Programming support, including (just a few):**
  - Guards
  - Hash-cons and function memoization
  - Packages
  - Mutable State, stobjs, arrays, applicative hash tables, …

- **System-level:** Emacs support, books and certification, abbreviated printing, parallelism (ACL2(p)), …
META-THEORETIC REASONING (2)

ACL2 supports a notion of “evaluation”, together with this sort of *meta* theorem, directing the use of `fn` to transform terms that are calls of `nth` or of `foo`.

```
(defthm fn-correct-1
  (equal (evl x a)
         (evl (fn x) a))
  :rule-classes ((:meta :trigger-fns (nth foo))))
```

More complex forms are supported, including:

- **extended-metametafunctions** that take `STATE` and contextual inputs;
- **transformations at the goal level**; and
- **hypotheses that extract known information** from the logical world.

For details, including issues pertaining to evaluation, see the *Essay on Correctness of Meta Reasoning* comment in the ACL2 sources. *Attachments* provide a challenge.
On Efficient Execution

Efficient execution is a key design goal.

- ACL2 definitions are actually programs in the Common Lisp programming language.
- Guards specify intended domains of functions and support sound, efficient Common Lisp evaluation.
- Several features support efficient computation by reusing storage, yet with a first-order logic foundation:
  - Single-threaded objects including state
  - Arrays
  - Function memoization (reuse of saved results)
  - Fast alists (applicative hash tables)