ACL2: Implementation of a Computational Logic

Matt Kaufmann
The University of Texas at Austin
Dept. of Computer Science
matthew.j.kaufmann@gmail.com

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

May 28, 2019 (Draft of April 1, 2019)

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

Now I maintain a computer program, ACL2, that proves theorems.
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....

Now I maintain a computer program, ACL2, that proves theorems.

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

Now I maintain a computer program, ACL2, that proves theorems.

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

**MY ANSWER**: Discuss the foundations of ACL2 as a practical application of mathematical logic.
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....

Now I maintain a computer program, ACL2, that proves theorems.

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

**MY ANSWER:** Discuss the foundations of ACL2 as a practical application of mathematical logic.

Please feel free to ask questions (in person or via email; I’ll put contact info and a link to the slides on the last slide).
Outline

Overview and Context

ACL2 Introduction

Logical Foundations for ACL2

Conclusion
OUTLINE

Overview and Context

ACL2 Introduction

Logical Foundations for ACL2

Conclusion
Overview and Context

The ACL2 home page has many useful links, and 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”.

The **ACL2 home page** has many useful links, and 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 (1)**

*Formal verification* (FV) of hardware and software systems is the use of tools that check correctness using mathematical methods, notably *proof*. 
**FORMAL VERIFICATION (1)**

*Formal verification* (FV) of hardware and software systems is the use of tools that check 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**.
**FORMAL VERIFICATION (1)**

*Formal verification* (FV) of hardware and software systems is the use of tools that check 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**.

As far as I know, ACL2 is the only ITP that has been used not only at universities and the U.S. Government, but also at several companies:
**Formal Verification (1)**

*Formal verification* (FV) of hardware and software systems is the use of tools that check 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**.

As far as I know, ACL2 is the only ITP that has been used not only at universities and the U.S. Government, but also at several companies:

- AMD, ARM, ArterisIP, Battelle, Centaur, GE, IBM, Intel, NXP, Kestrel, Oracle, Rockwell Collins
Two recent examples of ACL2 formalizations at UT Austin:

- **x86 interpreter**: models state transitions for x86 instructions
  - Testing validates faithfulness of this model to actual Intel x86 chips when running x86 machine code (approximately 3.3 million instructions per second).
  - Some x86 machine code programs have been proved correct.
  - It is under continued development at Centaur and Kestrel.

- **an efficient checker** for Boolean satisfiability (SAT) proofs
  - Used in recent international SAT competitions

---

**FORMAL VERIFICATION (2)**
In the Yearly ITP conference, many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems. ITP is typically more scalable than automatic theorem proving, but requires some human assistance. In ACL2, one proves lemmas that may be used automatically to simplify terms in later proofs. Some particular strengths of ACL2 among ITPs:

- Proof automation
- Proof debugging utilities
- Fast execution of programs
- Documentation (about 120,000 lines for just the system; many more for the libraries)
INTERACTIVE THEOREM PROVING

- Yearly ITP conference
Interactive Theorem Proving

- Yearly ITP conference
- Many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems.
Interactive Theorem Proving

- Yearly ITP conference
- Many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems.
- ITP is typically more scalable than automatic theorem proving, but requires some human assistance.
INTERACTIVE THEOREM PROVING

▶ Yearly ITP conference
▶ Many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems.
▶ ITP is typically more scalable than automatic theorem proving, but requires some human assistance.
  ▶ In ACL2, one proves lemmas that may be used automatically to simplify terms in later proofs.
INTERACTIVE THEOREM PROVING

- Yearly ITP conference
- Many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems.
- ITP is typically more scalable than automatic theorem proving, but requires some human assistance.
  - In ACL2, one proves lemmas that may be used automatically to simplify terms in later proofs.

Some particular strengths of ACL2 among ITPs:
Interactive Theorem Proving

- Yearly **ITP conference**
- Many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems.
- ITP is typically more scalable than automatic theorem proving, but requires some human assistance.
  - **In ACL2**, one proves lemmas that may be used automatically to simplify terms in later proofs.

Some particular strengths of ACL2 among ITPs:

- Proof automation
INTERACTIVE THEOREM PROVING

- Yearly ITP conference
- Many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems.
- ITP is typically more scalable than automatic theorem proving, but requires some human assistance.
  - In ACL2, one proves lemmas that may be used automatically to simplify terms in later proofs.

Some particular strengths of ACL2 among ITPs:

- Proof automation
- Proof debugging utilities
Interactive Theorem Proving

- Yearly ITP conference
- Many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems.
- ITP is typically more scalable than automatic theorem proving, but requires some human assistance.
  - In ACL2, one proves lemmas that may be used automatically to simplify terms in later proofs.

Some particular strengths of ACL2 among ITPs:

- Proof automation
- Proof **debugging** utilities
- Fast execution of programs
Interactive Theorem Proving

- Yearly ITP conference
- Many ITP systems (e.g., ACL2) can send sub-problems to automatic proof tools, e.g., SAT solvers for Boolean problems.
- ITP is typically more scalable than automatic theorem proving, but requires some human assistance.
  - In ACL2, one proves lemmas that may be used automatically to simplify terms in later proofs.

Some particular strengths of ACL2 among ITPs:

- Proof automation
- Proof debugging utilities
- Fast execution of programs
- Documentation (about 120,000 lines for just the system; many more for the libraries)
OUTLINE

Overview and Context

ACL2 Introduction

Logical Foundations for ACL2

Conclusion
Outline

Overview and Context

ACL2 Introduction

Logical Foundations for ACL2

Conclusion
ACL2 Introduction

- Freely available, including libraries of *certifiable books*
ACL2 Introduction

- Freely available, including libraries of certifiable books
- Let's look at the ACL2 home page.
ACL2 Introduction

- Freely available, including libraries of certifiable books
- Let’s look at the ACL2 home page.
- ACL2 is written mostly in itself (!).
ACL2 Introduction

- Freely available, including libraries of *certifiable books*
- Let’s look at the ACL2 home page.
- ACL2 is written mostly in itself (!).
  - About 11 MB of source code (including comments but not including documentation)
ACL2 INTRODUCTION

- Freely available, including libraries of certifiable books
- Let’s look at the ACL2 home page.
- ACL2 is written mostly in itself (!).
  - About 11 MB of source code (including comments but not including documentation)
- The ACL2 system and its libraries (community books) are available from the ACL2 home page and from Github.
ACL2 Introduction

- Freely available, including libraries of *certifiable books*
- Let’s look at the ACL2 home page.
- ACL2 is written mostly in itself (!).
  - About 11 MB of source code (including comments but not including documentation)
- The ACL2 system and its libraries (*community books*) are available from the ACL2 home page and from Github.
  - More than 500,000 *events* (theorems, definitions, other) are evaluated in the community books.
ACL2 Introduction

- Freely available, including libraries of certifiable books
- Let’s look at the ACL2 home page.
- ACL2 is written mostly in itself (!).
  - About 11 MB of source code (including comments but not including documentation)
- The ACL2 system and its libraries (community books) are available from the ACL2 home page and from Github.
  - More than 500,000 events (theorems, definitions, other) are evaluated in the community books.
- Workshops: Latest (#15) was at UT Austin, Nov. 5-6, 2018.
ACL2 Introduction

- Freely available, including libraries of certifiable books
- Let’s look at the ACL2 home page.
- ACL2 is written mostly in itself (!).
  - About 11 MB of source code (including comments but not including documentation)
- The ACL2 system and its libraries (community books) are available from the ACL2 home page and from Github.
  - More than 500,000 events (theorems, definitions, other) are evaluated in the community books.
- Workshops: Latest (#15) was at UT Austin, Nov. 5-6, 2018.
- History
ACL2 INTRODUCTION

» Freely available, including libraries of certifiable books
» Let’s look at the ACL2 home page.
» ACL2 is written mostly in itself (!).
  » About 11 MB of source code (including comments but not including documentation)
» The ACL2 system and its libraries (community books) are available from the ACL2 home page and from Github.
  » More than 500,000 events (theorems, definitions, other) are evaluated in the community books.
» Workshops: Latest (#15) was at UT Austin, Nov. 5-6, 2018.
» History
ACL2 Introduction

- Freely available, including libraries of certifiable books
- Let’s look at the ACL2 home page.
- ACL2 is written mostly in itself (!).
  - About 11 MB of source code (including comments but not including documentation)
- The ACL2 system and its libraries (community books) are available from the ACL2 home page and from Github.
  - More than 500,000 events (theorems, definitions, other) are evaluated in the community books.
- Workshops: Latest (#15) was at UT Austin, Nov. 5-6, 2018.
- History
  - Boyer-Moore Theorem Provers go back to their collaboration starting in 1971.
USING ACL2

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

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

- **ACL2 as an automatic theorem prover**
  
  [DEMO]: file `demo-2.lsp`
  (log `demo-2-log.txt`)
USING ACL2

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

» ACL2 as an automatic 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 provable, then that goal is provable.
USING ACL2

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

- ACL2 as an automatic 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 provable, then that goal is provable.

- Interfaces include Emacs, ACL2 Sedan (Eclipse-based), none.
A longer talk on ACL2, oriented towards CS graduate students and with more focus on using ACL2, is here:

Using ACL2 (2)

A longer talk on ACL2, oriented towards CS graduate students and with more focus on using ACL2, is here:


That talk mentions this link to several demos and their logs:

PARTIAL TIMELINE

- 1970: Boyer and Moore meet
- 1975: Insertion sort, binary adder, expression compiler, prime factorization, BDX930 abandoned
- 1980: RSA, unsolvability of halting problem
- 1985: FM8502, Gauss, Unity, Piton, KIT OS kernel, micro Gypsy compiler, clock sync
- 1990: FM8501, Gödel, FM9001, Byzantine Generals, Motorola 68020, biphase mark
- 2000: AMD K5 floating-point division µcode, real-time model, Rockwell JEM1
- 2005: Unity, Galois/Rockwell SHADE, fast consensus analysis
- 2010: Y86, Y86 with STOBJ, Dijkstra shortest path, Kalman filters
- 2015: X86 ISA, UCLID integration prototype, AAMP7G MIL cert.
Overview and Context

ACL2 Introduction

Logical Foundations for ACL2

Conclusion
OUTLINE

Overview and Context

ACL2 Introduction

Logical Foundations for ACL2

Conclusion
Logical Foundations (1)

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

I suspect that weaker induction would usually suffice in practice;
maybe only $\omega^\omega$;
maybe only to each of $\omega, \omega^\omega, \omega^\omega^\omega$, etc., iterated through only standard natural numbers …
The ACL2 logic is a first-order logic with induction up to $\varepsilon_0$.

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

► . . . but it hasn’t been a priority to work this out, let alone consider effects on the implementation.
Logical Foundations (2)

Restriction: All ACL2 theories extend a given ground-zero theory, which is essentially Peano Arithmetic with $\varepsilon_0$-induction, extended with data types for:

- numbers (complex rationals),
- characters,
- strings,
- symbols, and
- closure under a pairing operation (cons).

This gives us lists, where the symbol nil represents the empty list. For example:

ACL2 !>(cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
ACL2 !>
Logical Foundations (2)

Restriction: All ACL2 theories extend a given ground-zero theory, which is essentially Peano Arithmetic with $\varepsilon_0$-induction, extended with data types for:

- numbers (complex rationals),
- characters,
- strings,
- symbols, and
- closure under a pairing operation (cons).
**Logical Foundations (2)**

*Restriction:* All ACL2 theories extend a given *ground-zero* theory, which is essentially Peano Arithmetic with \( \varepsilon_0 \)-induction, extended with data types for:

- numbers (complex rationals),
- characters,
- strings,
- symbols, and
- closure under a pairing operation (*cons*).

This gives us lists, where the symbol *nil* represents the empty list. For example:

ACL2 !>(cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
ACL2 !>
Logical Foundations (3)

ACL2 extensions are generally conservative (no new theorems in the existing language).
ACL2 extensions are generally conservative (no new theorems in the existing language).

- This holds even for recursive definitions, since “termination” must be provable.
Logical Foundations (3)

ACL2 extensions are generally *conservative* (no new theorems in the existing language).

- ... This holds even for recursive definitions, since “termination” must be provable.
Logical Foundations (3)

ACL2 extensions are generally conservative (no new theorems in the existing language).

- ... This holds even for recursive definitions, since "termination" must be provable.
- So, one may introduce new concepts *locally* when carrying out proofs.
EXTENSION PRINCIPLE: DEFINITIONS

A definition extends the current theory with the axiom equating the call with the body.
EXTENSION PRINCIPLE: DEFINITIONS

A definition extends the *current theory* with the axiom equating the call with the body. **Example:**

```
(defun rev (x)
  (if (consp x)
      (append (rev (rest x))
              (cons (first x) nil))
     nil))
```
EXTENSION PRINCIPLE: DEFINITIONS

A definition extends the current theory with the axiom equating the call with the body. **Example:**

(defun rev (x)
  (if (consp x)
      (append (rev (rest x))
              (cons (first x) nil))
     nil))

The axiom added is (the universal closure of):

(rev x) =
(if (consp x)
  (append (rev (rest x))
          (cons (first x) nil))
 nil)
EXTENSION PRINCIPLE: DEFINITIONS

A definition extends the current theory with the axiom equating the call with the body. **Example:**

```
(defun rev (x)
  (if (consp x)
      (append (rev (rest x))
              (cons (first x) nil))
      nil))
```

The axiom added is (the universal closure of):

```
(rev x) =
(if (consp x)
    (append (rev (rest x))
            (cons (first x) nil))
    nil)
```

The definition may be recursive if some measure into $\varepsilon_0$ is proved to decrease on each recursive call.
**Extension Principle: Choice (and ∃)**

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.
**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 \(P(\vec{x})\):

\[
\begin{align*}
    w(\vec{x}) &= \varepsilon \vec{y} A(\vec{x}, \vec{y}) \\
    P(\vec{x}) &= A(\vec{x}, w(\vec{x}))
\end{align*}
\]
**EXTENSION PRINCIPLE: CHOICE (AND ∃)**

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 \( P(\vec{x}) \):

\[ w(\vec{x}) = \varepsilon \vec{y} A(\vec{x}, \vec{y}) \]

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

```lisp
(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 ∃) (2)**

This sort of thing is clearly conservative (assuming the Axiom of Choice or at least well-orderable models). . .
**Extension Principle: Choice (and ∃) (2)**

This sort of thing is clearly conservative (assuming the Axiom of Choice or at least well-orderable models) . . .

. . . IF we ignore induction!
This sort of thing is clearly conservative (assuming the Axiom of Choice or at least well-orderable models). . .

. . . IF we ignore induction!

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

It is also legal to introduce *constrained* functions, using axioms that are *proved* about *local witnesses*.
EXTENSION PRINCIPLE: CONSTRAINTS

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

**Example:**

```lisp
(encapsulate ( ((fn * *) => *))
  (local (defun fn (x y)
        (+ x y)))
  (defthm fn-commutative
    (equal (fn x y) (fn y x))))
```
**Extension principle: Constraints**

It is also legal to introduce *constrained* functions, using axioms that are *proved* about *local witnesses*.

**Example:**

```lisp
(encapsulate ( ((fn * *) => *) )
  (local (defun fn (x y)
             (+ x y)))
  (defthm fn-commutative
             (equal (fn x y) (fn y x))))
```

A derived inference rule, *functional instantiation*, is often useful with constrained functions. The next slide shows an 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-rev
  (implies (equal (len lst1) (len lst2))
           (equal (map2-fn (rev lst1) (rev lst2))
                  (rev (map2-fn lst1 lst2)))))
(defun map2-fn (lst1 lst2)
  (if (consp lst1)
      (cons (fn (first lst1) (first lst2))
        (map2-fn (rest lst1) (rest lst2)))
      nil))

(defthm map2-fn-rev
  (implies (equal (len lst1) (len lst2))
    (equal (map2-fn (rev lst1) (rev lst2))
      (rev (map2-fn lst1 lst2))))))

(defun map2-* (lst1 lst2)
  (if (consp lst1)
      (cons (* (first lst1) (first lst2))
        (map2-* (rest lst1) (rest lst2)))
      nil))

(defthm map2-*-rev
  (implies (equal (len lst1) (len lst2))
    (equal (map2-* (rev lst1) (rev lst2))
      (rev (map2-* lst1 lst2))))))

:hints ("Goal" :by (:functional-instance
                      map2-fn-rev
                      (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:
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.
**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 $n$. 

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: Clean formalization (which I’ll flash on the next slide)

25 lines
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: Clean formalization (which I’ll flash on the next slide)
  25 lines
- overspill-proof.lisp: Ugly proof, but LOCAL to the main proof, by conservativity
  256 lines
Key parts of the book *overspill.lisp*:

(local (include-book "overspill-proof"))
(set-enforce-redundancy t)
(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:
Meta-theoretic Reasoning (1)

In ACL2, you can:

- code a simplifier,
META-THEORETIC REASONING (1)

In ACL2, you can:

- code a simplifier,
- prove that it is sound, and
Meta-theoretic Reasoning (1)

In ACL2, you can:

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

Efficient execution can be important for meta-theoretic reasoning!
Meta-theoretic Reasoning (1)

In ACL2, you can:

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

Efficient execution can be important for meta-theoretic reasoning!
We can return to this on an extra slide, if there is time and interest.
Other Logical Challenges

Here are some other challenges in the foundations of ACL2.
Other Logical Challenges

Here are some other challenges in the foundations of ACL2.

- Packages provide namespaces — e.g., `PKG1::F` and `PKG2::F` are distinct. But packages introduce axioms such as `symbol-package-name(PKG1::F) = "PKG1"`. So package introduction is not conservative and hence must be recorded.
Other Logical Challenges

Here are some other challenges in the foundations of ACL2.

- **Packages** provide namespaces — e.g., `PKG1 : F` and `PKG2 : F` are distinct. But packages introduce axioms such as `symbol-package-name(PKG1 : F) = "PKG1"`. So package introduction is *not conservative* and hence 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**?
**Other Logical Challenges**

Here are some other challenges in the foundations of ACL2.

- **Packages** provide namespaces — e.g., `PKG1 : : F` and `PKG2 : : F` are distinct. But packages introduce axioms such as `symbol-package-name(PKG1 : : F) = "PKG1"`. So package introduction is *not conservative* and hence 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.
Defattach allows extensions that are not conservative.

Example:
**DEFATTACH (1)**

Defattach allows extensions that are **not** conservative.

**Example:**

- **Constraint for** a “specification” function, $\text{spec}$:
  \[ x \in \mathbb{Z} \implies \text{spec}(x) \in \mathbb{Z} \]
**Defattach (1)**

Defattach 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`:
  \[ f(x, y) = \text{spec}(x + y) \]
Defattach (1)

Defattach allows extensions that are not conservative.

Example:

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

- **Define** function `f`: `f(x, y) = spec(x + y)`

- **Define** an “implementation” function, `impl`:
  \[ impl(x) = 10 \times x \]
**Defattach** (1)

**Defattach** 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:
  \[ \text{impl}(x) = 10 \times x \]

- **Attach** impl to spec:
  \( \text{(defattach spec impl)} \)
Defattach allows extensions that are not conservative.

Example:

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

- **Define** function `f`: \[ f(x, y) = spec(x + y) \]

- **Define** an “implementation” function, `impl`:
  \[ impl(x) = 10 \times x \]

- **Attach** `impl` to `spec`:
  \[ \text{(defattach spec impl)} \]

Result not provable from axioms for `f` and `spec`:

ACL2 !>(f 3 4) ; = spec(7)
**Defattach (1)**

*Defattach* 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*:
  \[ \text{impl}(x) = 10 \times x \]

- **Attach** *impl* to *spec*:
  \[
  \text{(defattach spec impl)}
  \]

**Result not provable from axioms for *f* and *spec*:**

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

Issues to consider:
DEFATTACH (2)

Issues to consider:

▶ Is (local (defattach ...)) supported?
DEFATTACH (2)

Issues to consider:

- Is (local (defattach ...)) supported?
  YES, local is supported.
Defattach (2)

Issues to consider:

- Is `(local (defattach ...) )` supported?
  - Yes, `local` is supported.

- Then how do we deal with conservativity?
DEFATTACH (2)

Issues to consider:

▶ **Is** \((\text{local} \ (\text{defattach} \ ...))\) supported?  
**YES**, \text{local} is supported.

▶ **Then how do we deal with conservativity?**  
\textbf{Two theories}: The \textit{current theory} for reasoning and a \textit{stronger evaluation theory}, extended using \texttt{defattach}:

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

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

- **Ah, but what about this?**

  `(thm (equal (f 3 4) 70))`
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`:

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

▶ Ah, but what about this?

(thm (equal (f 3 4) 70))

The proof fails! (Good!)
**DEFATTACH (2)**

**Issues to consider:**

- **Is** \((\text{local } (\text{defattach } \ldots))\) **supported?**
  
  \textbf{YES, local is supported.}

- **Then how do we deal with conservativity?**
  
  \textbf{Two theories:} The \textit{current theory} for reasoning and a \textit{stronger evaluation theory}, extended using \texttt{defattach}:

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

- **Ah, but what about this?**

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

  \textbf{The proof fails! (Good!)}

- **Why is the evaluation theory consistent?**
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:
  \[ \text{spec}(x) = \text{impl}(x) \]

▶ Ah, but what about this?
  \[ \text{(thm (equal (f 3 4) 70))} \]
  The proof fails! (Good!)

▶ Why is the evaluation theory consistent?
  A key requirement is that the attachment relation is suitably acyclic.
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:
   \[
   \text{spec}(x) = \text{impl}(x)
   \]

▶ Ah, but what about this?
   (thm (equal (f 3 4) 70))
   The proof fails! (Good!)

▶ Why is the evaluation theory consistent?
   A key requirement is that the attachment relation is suitably acyclic.

For details, including issues pertaining to evaluation, see the Essay on Defattach comment in the ACL2 sources.
"HIGHER-ORDER" \texttt{Apply$^\dagger$} (1)

One application of \texttt{defattach} is a mechanism for applying function symbols.
“**HIGHER-ORDER**" Apply\$ (1)

One application of defattach is a mechanism for applying function symbols. **Example:**

```
(include-book "projects/apply/top" :dir :system)
(defun$ norm^2 (x)
  (+ (* (car x) (car x)) (* (cdr x) (cdr x))))
(assert-event
  (equal (norm^2 (cons 3 4)) 25))
(thm (equal (norm^2 (cons 3 4)) 25))
(assert-event
  (equal (apply$ 'norm^2 (list (cons 3 4)))
        25))
```
"HIGHER-ORDER" Apply$ (1)

One application of defattach is a mechanism for applying function symbols. **Example:**

```
(defun norm^2 (x)
  (+ (* (car x) (car x)) (* (cdr x) (cdr x))))
```

```
(assert-event
  (equal (norm^2 (cons 3 4)) 25))
(thm (equal (norm^2 (cons 3 4)) 25))
```

```
(assert-event
  (equal (apply$ 'norm^2 (list (cons 3 4)))
         25))
```

But this fails!

```
(thm (equal (apply$ 'norm^2
                   (list (cons 3 4)))
           25))
```
"HIGHER-ORDER" Apply$: (2)

However, the proof succeeds for the `thm` below, where the `warrant hypothesis`, `(warrant norm^2), asserts:

`(∀ x) (equal (apply$ 'norm^2 (list x))
(norm^2 x))."
“HIGHER-ORDER” Apply$ (2)

However, the proof succeeds for the thm below, where the warrant hypothesis, (warrant norm^2), asserts:

(∀ x) (equal (apply$ 'norm^2 (list x)) (norm^2 x)).

(thm (implies (warrant norm^2)
 (equal (apply$ 'norm^2 (list (cons 3 4)))
 25)))
“HIGHER-ORDER” Apply$\ (2)

However, the proof succeeds for the thm below, where the 
**warrant hypothesis**, (warrant norm^2), **asserts:**
(∀ x) (equal (apply$ 'norm^2 (list x))
  (norm^2 x)).

(thm (implies (warrant norm^2)
  (equal (apply$ 'norm^2
     (list (cons 3 4)))
   25)))

Warrant hypotheses are not vacuous!
We show there is a natural *evaluation theory* where every 
warrant is attached to the constant “true” function.
ITERATION

A key application of apply$: iteration, which is useful for programming, and has reasoning support in ACL2. **Example:**

```
ACL2 !>(loop$ for i from 1 to 4 sum (* i i))
30
ACL2 !>
```
ITERATION

A key application of apply$ is: *iteration*, which is useful for programming, and has reasoning support in ACL2. **Example:**

ACL2 !>(loop$ for i from 1 to 4 sum (* i i))
30
ACL2 !>

ACL2 treats this essentially as follows

(SUM$ 'LAMBDA (I) (* I I))
  (FROM-TO-BY 1 4 1))
ITERATION

A key application of apply$: iteration, which is useful for programming, and has reasoning support in ACL2. **Example:**

ACL2 !>(loop$ for i from 1 to 4 sum (* i i))
30
ACL2 !>

ACL2 treats this essentially as follows

(SUM$ '(LAMBDA (I) (* I I))
   (FROM-TO-BY 1 4 1))

where sum$ is defined essentially as follows.

(defun sum$ (fn lst)
   (if (endp lst)
      0
      (+ (apply$ fn (list (car lst)))
         (sum$ fn (cdr lst))))
Overview and Context

ACL2 Introduction

Logical Foundations for ACL2

Conclusion
OUTLINE

Overview and Context

ACL2 Introduction

Logical Foundations for ACL2

Conclusion
CONCLUSION

- ACL2 has a 29 (or 48) year history and is used in industry.

— Anna Slobodova, verification manager at Centaur Technology

As an ITP system, it relies on user guidance for large problems but enjoys scalability.

Mechanizing a logic, for efficient and flexible evaluation and proof, can present challenges.

For more information, see the ACL2 home page, in particular links to The Tours and Publications, which links to introductory material.
CONCLUSION

- ACL2 has a 29 (or 48) year history and is used in industry.
  - People are actually *paid* to prove theorems with ACL2.
CONCLUSION

- ACL2 has a 29 (or 48) year history and is used in industry.
  - 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 at Centaur Technology
CONCLUSION

- ACL2 has a 29 (or 48) year history and is used in industry.
  - 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 at Centaur Technology

- As an ITP system, it relies on user guidance for large problems but enjoys scalability.
CONCLUSION

- ACL2 has a 29 (or 48) year history and is used in industry.
  - 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 at Centaur Technology

- As an ITP system, it relies on user guidance for large problems but enjoys scalability.

- Mechanizing a logic, for efficient and flexible evaluation and proof, can present challenges.
CONCLUSION

- ACL2 has a 29 (or 48) year history and is used in industry.
  - 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 at Centaur Technology

- As an ITP system, it relies on user guidance for large problems but enjoys scalability.

- Mechanizing a logic, for efficient and flexible evaluation and proof, can present challenges.

- For more information, see the ACL2 home page, in particular links to The Tours and Publications, which links to introductory material.
THANK YOU!
THANK YOU!

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

```lisp
(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-metafunctions 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.
**Meta-theoretic Reasoning (2)**

ACL2 supports a notion of “eval”, 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 (list (:meta :trigger-fns (nth foo))))
```
META-THEORETIC REASONING (2)

ACL2 supports a notion of “eval”, 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:
META-THEORETIC REASONING (2)

ACL2 supports a notion of “eval”, together with this sort of meta theorem, directing the use of \texttt{fn} to transform terms that are calls of \texttt{nth} or of \texttt{foo}.

\begin{verbatim}
(defthm fn-correct-1
  (equal (evl x a)
         (evl (fn x) a))
  :rule-classes ((:meta :trigger-fns (nth foo))))
\end{verbatim}

More complex forms are supported, including:

- \texttt{extended-metametafunctions} that take \texttt{STATE} and contextual inputs;
META-THEORETIC REASONING (2)

ACL2 supports a notion of “eval”, 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 (eval x a)
         (eval (fn x) a))
  :rule-classes ((:meta :trigger-fns (nth foo))))
```

More complex forms are supported, including:

- **extended-metafunctions** that take `STATE` and contextual inputs;
- **transformations at the goal level**; and
**META-THEORETIC REASONING (2)**

ACL2 supports a notion of “eval”, 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-metafunctions** that take `STATE` and contextual inputs;
- **transformations at the goal level**; and
- **hypotheses** that extract known information from the logical world.
**META-THEORETIC REASONING (2)**

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

```lisp
(defun 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-metastatements** 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.
Meta-theoretic Reasoning (2)

ACL2 supports a notion of “eval”, 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-metafunctions 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.
ON EFFICIENT EXECUTION

*Efficient execution* is a key design goal.

- ACL2 definitions are actually programs in the Common Lisp programming language.
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.
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)