ACL2:
A Program Verifier
for
Applicative Common Lisp

Matt Kaufmann - AMD, Austin, TX
J Strother Moore – UT CS Dept, Austin, TX

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*

A Sample Lisp Program

```
(defun append (x y)
  (if (endp x)
      y
      (cons (car x)
            (append (cdr x) y))))

(append '(1 2 3) (append '(4 5 6) '(7 8 9)))
= '(1 2 3 4 5 6 7 8 9)
```

A Conjectured Property

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Turning (Applicative) Lisp into a Mathematical Logic

Axioms:

$\texttt{t} \neq \texttt{nil}$

$\texttt{x} = \texttt{nil} \rightarrow \texttt{(if x y z)} = \texttt{z}$

$\texttt{x} \neq \texttt{nil} \rightarrow \texttt{(if x y z)} = \texttt{y}$

```
(car (cons x y)) = x

(cdr (cons x y)) = y

(endp nil) = t

(endp (cons x y)) = nil
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (append b c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (append b c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Base Case:  (endp a).
(equal (append b c)
       (append b c))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (append b c)
       (append b c))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: (endp a).

T

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Induction Step: (not (endp a)).
(equal (append (append a b) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))

Proof: by induction on a.

Induction Step: (not (endp a)).
(equal (append (cons (car a)
                     (append (cdr a) b)) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (append (cons (car a)
                     (append (cdr a) b)) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
             (append (append (cdr a) b) c))
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
         (append (append (cdr a) b) c))
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))

Proof: by induction on a.

Induction Step: (not (endp a)).
(equal (cons (car a)
         (append (append (cdr a) b) c))
       (cons (car a)
         (append (cdr a) (append b c))))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Induction Step: (not (endp a)).
(equal (cons (car a)
         (append (append (cdr a) b) c))
       (cons (car a)
         (append (cdr a) (append b c))))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal
         (append (append (cdr a) b) c)

         (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Induction Step: (not (endp a)).
(equal (append (append (cdr a) b) c)
       (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (append (append (cdr a) b) c)
       (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

T

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Q.E.D.

Outline of This Talk

- What is ACL2?
- How does it work?
- Is it Useful?
- Why?

A **C**omputational **L**ogic for
Applicative **C**ommon **L**isp

= **ACL2**

- a functional subset of Common Lisp
- a first-order mathematical logic
- a proof-construction assistant/environment

Lisp's Implicit Typing

We proved

```
(equal (append (append a b) c)
       (append a (append b c)))
```

So this is supposed to evaluate to true *for all* values of `a`, `b`, and `c`?

But

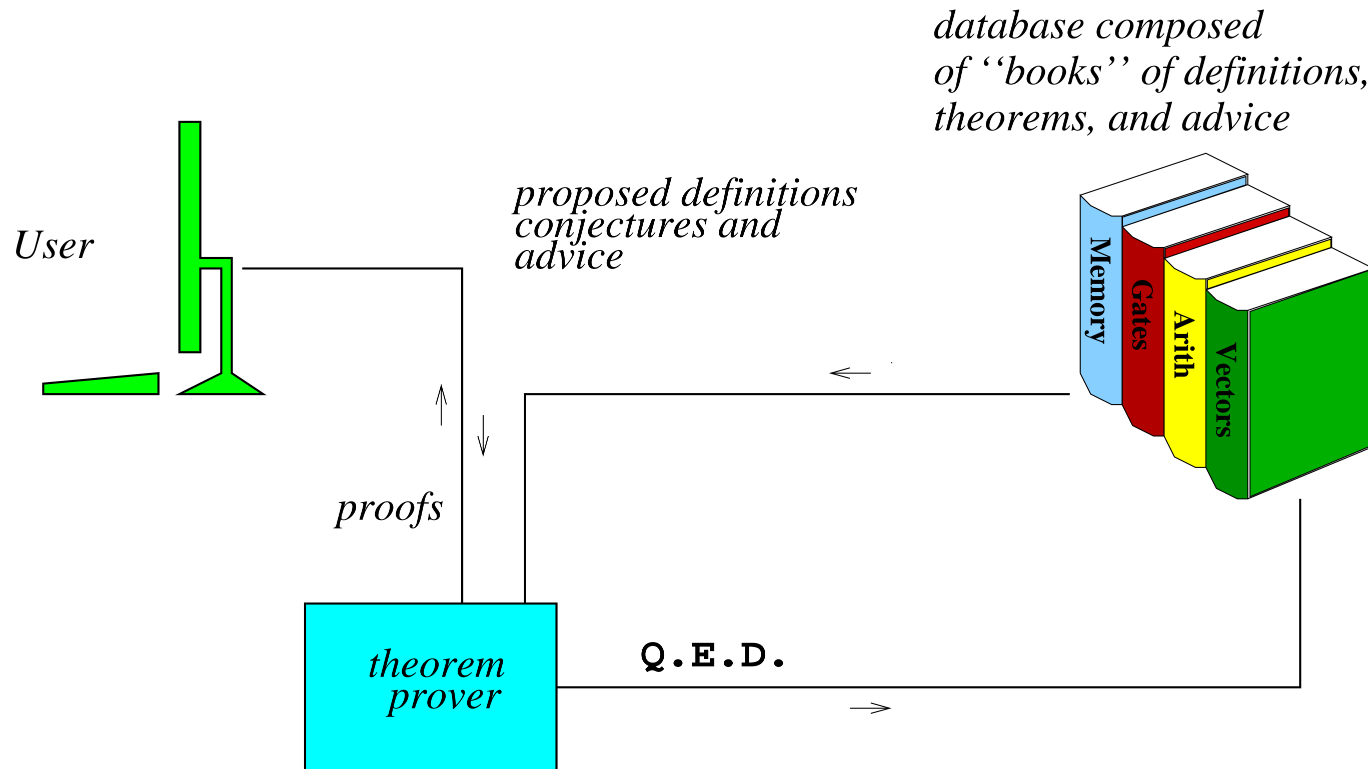
```
? (let ((a '(1 2 3 . 4))
        (b '(5 6 7))
        (c '(8 9 10)))
    (equal (append (append a b) c)
           (append a (append b c))))

Error in process listener(2): value 4
is not of the expected type LIST.
```

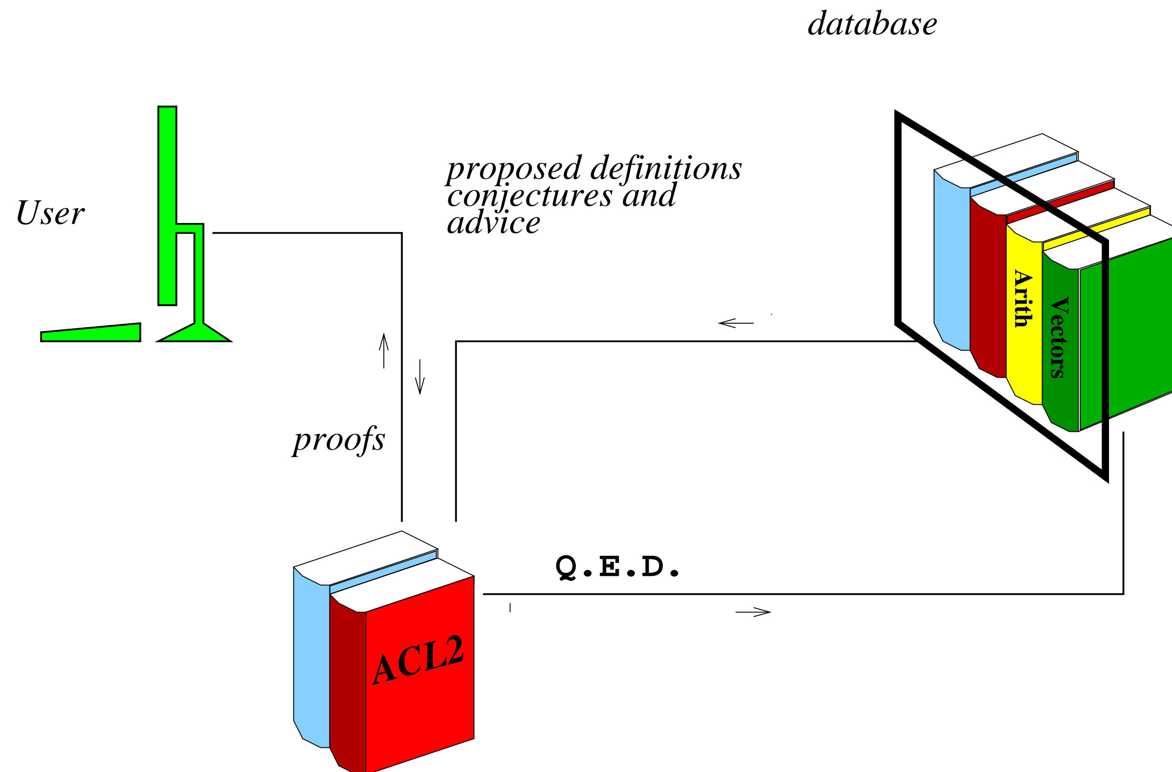
Our paper explains how we classify some theorems as *Common Lisp compliant*.

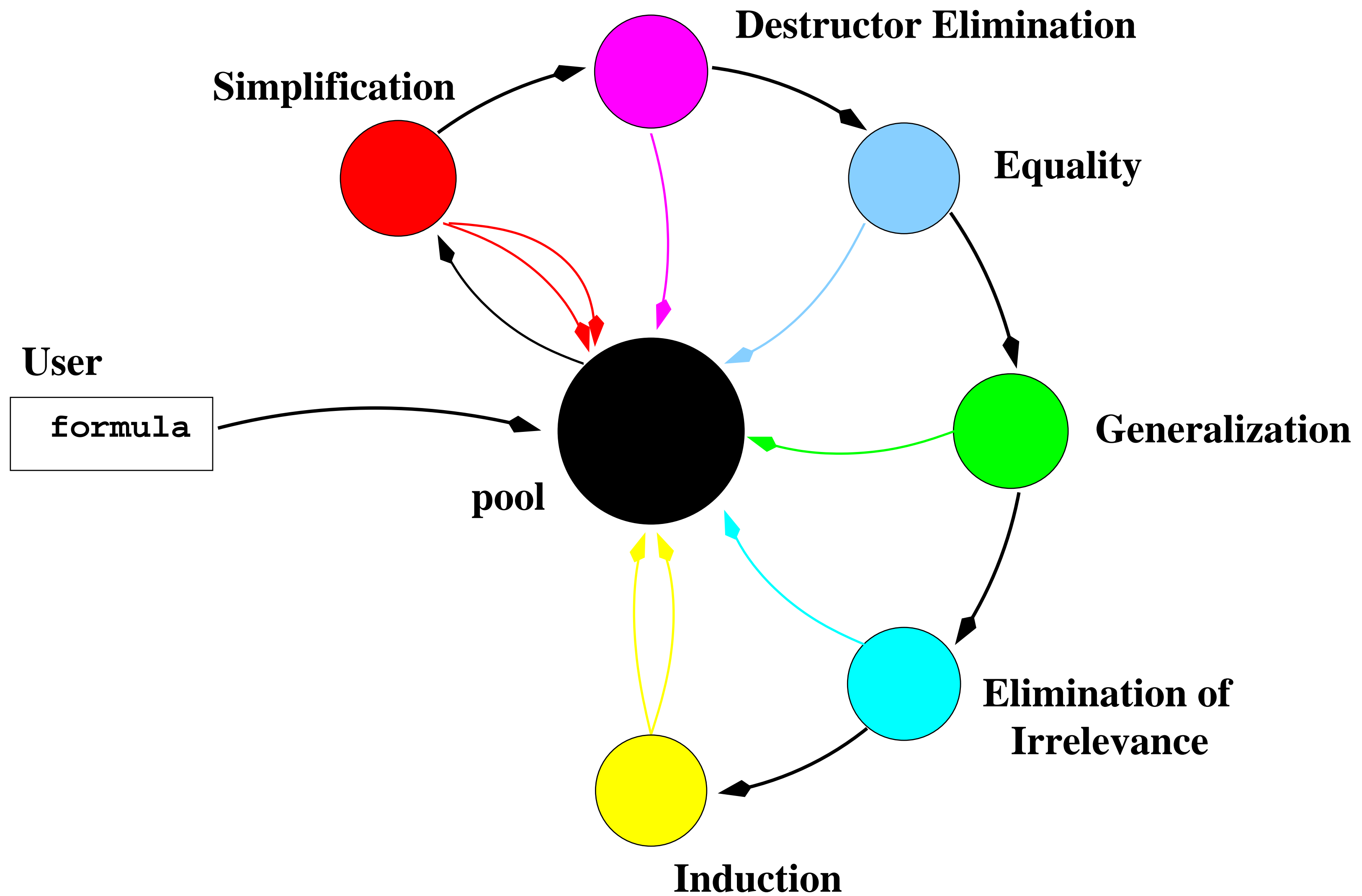
We don't deal with this in this talk, but ACL2 allows the user to declare the expected types of arguments and prove that they are respected.

How does the theorem prover work?



How does the theorem prover work?




Destructor Elimination
Simplification
Equality
User
formula
Generalization
pool
Elimination of
Irrelevance
Induction

Simplification

- congruence-based conditional rewriting
- use of recursively defined function definitions
- use of previously proved theorems as rules
- efficient representation of constants

- very efficient evaluation
- fixed-point based typing procedure
- back- and forward-chaining
- integrated linear arithmetic decision procedure
- extensions supporting non-linear arithmetic

- integrated equality decision procedure
- integrated BDD decision procedure
- connected SAT decision procedure (in progress)
- reflection (metafunctions)
- single-threaded objects ("monads")

- beta-reduction avoidance
- hint mechanism for steering automatic features
- interactive proof-checker for user control
- proof trees display for proof navigation
- decades of engineering

ACL2 Demo 1

Is it useful?

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," Nov 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, Jan 23, 1995*

IEEE 754 Floating Point Standard

Elementary operations are to be performed as though the infinitely precise (standard mathematical) operation were performed and then the result rounded to the indicated precision.

AMD K5 Algorithm FDIV($p, d, mode$)

1.	sd_0	$= \text{lookup}(d)$	[exact 17 8]
2.	d_r	$= d$	[away 17 32]
3.	sdd_0	$= sd_0 \times d_r$	[away 17 32]
4.	sd_1	$= sd_0 \times \text{comp}(sdd_0, 32)$	[trunc 17 32]
5.	sdd_1	$= sd_1 \times d_r$	[away 17 32]
6.	sd_2	$= sd_1 \times \text{comp}(sdd_1, 32)$	[trunc 17 32]
...	...	$= \ldots$	...
29.	q_3	$= sd_2 \times ph_3$	[trunc 17 24]
30.	qq_2	$= q_2 + q_3$	[sticky 17 64]
31.	qq_1	$= qq_2 + q_1$	[sticky 17 64]
32.	$fdiv$	$= qq_1 + q_0$	$mode$

Using the Reciprocal

```
            3 6.
  +          -.1 7
  +            .0 0 3 4
  +          - .0 0 0 0 6 6
          ---------------------
            3 5.8 3 3 3 3 4
          ---------------------
  1 2  /  4 3 0.0 0 0 0 0 0 0
          4 3 2.
         ---------
             -2.
             -2.0 4
            --------
               .0 4
               .0 4 0 8
              ----------
           -   .0 0 0 8
           -   .0 0 0 7 9 2
              --------------
           -   .0 0 0 0 0 8
```

Reciprocal Calculation:

$1/12 \;=\; 0.083\overline{3} \approx 0.083 = sd_2$

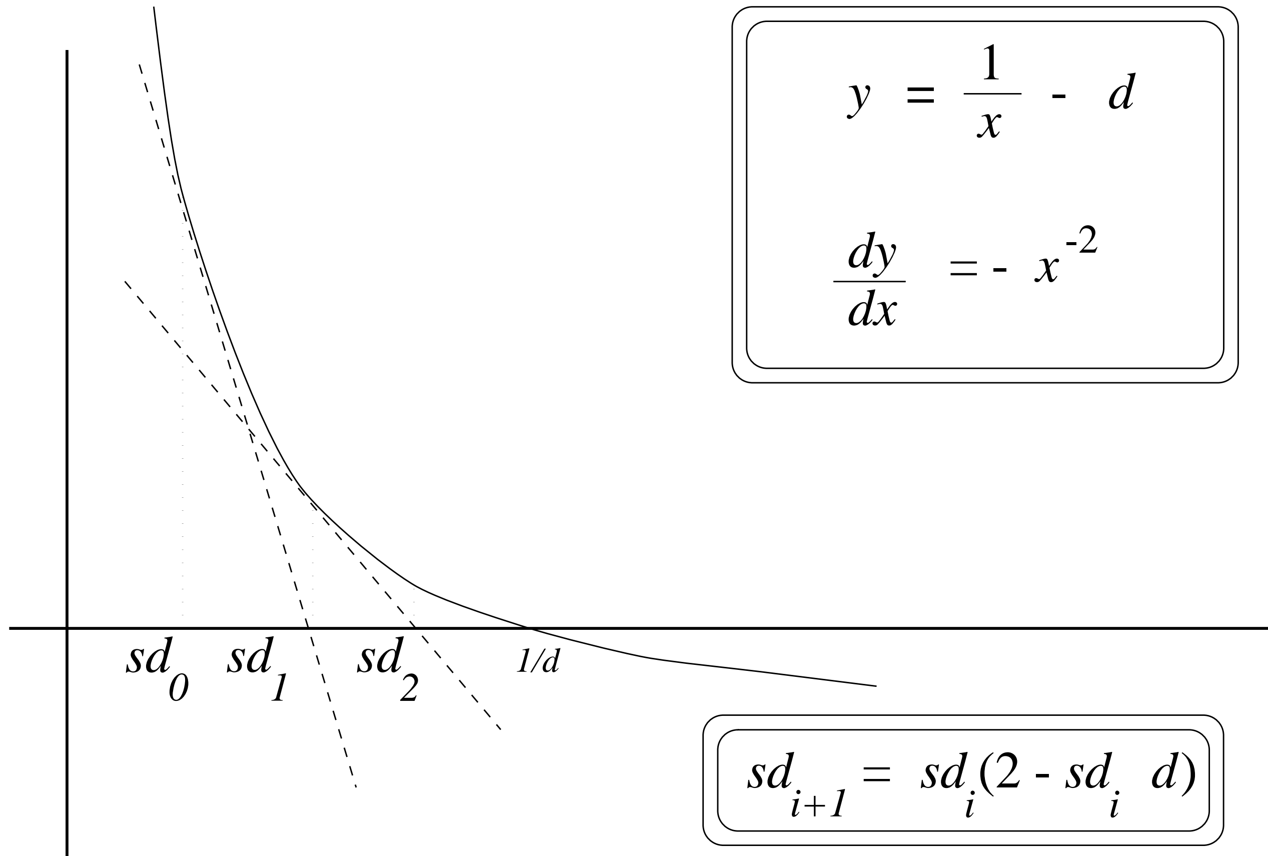
Quotient Digit Calculation:

$$0.083 \times 430.0000 = 35.6900000 \approx 36.000000 = q_0$$
$$0.083 \times -2.0000 = -.1660000 \approx -.170000 = q_1$$
$$0.083 \times .0400 = .0033200 \approx .003400 = q_2$$
$$0.083 \times -.0008 = -.0000664 \approx -.000067 = q_3$$

Summation of Quotient Digits:

$q_0 + q_1 + q_2 + q_3 = 35.833333$

Computing the Reciprocal



top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse
1.0000000_2	0.11111111_2	1.0100000_2	0.11001100_2	1.1000000_2	0.10101010_2	1.1100000_2	0.10010010_2
1.0000001_2	0.11111101_2	1.0100001_2	0.11001011_2	1.1000001_2	0.10101001_2	1.1100001_2	0.10010001_2
1.0000010_2	0.11111011_2	1.0100010_2	0.11001010_2	1.1000010_2	0.10101000_2	1.1100010_2	0.10010001_2
1.0000011_2	0.11111001_2	1.0100011_2	0.11001000_2	1.1000011_2	0.10101000_2	1.1100011_2	0.10010000_2
1.0000100_2	0.11110111_2	1.0100100_2	0.11000111_2	1.1000100_2	0.10100111_2	1.1100100_2	0.10001111_2
1.0000101_2	0.11110101_2	1.0100101_2	0.11000110_2	1.1000101_2	0.10100110_2	1.1100101_2	0.10001111_2
1.0000110_2	0.11110100_2	1.0100110_2	0.11000101_2	1.1000110_2	0.10100101_2	1.1100110_2	0.10001110_2
1.0000111_2	0.11110010_2	1.0100111_2	0.11000100_2	1.1000111_2	0.10100100_2	1.1100111_2	0.10001110_2
1.0001000_2	0.11110000_2	1.0101000_2	0.11000010_2	1.1001000_2	0.10100011_2	1.1101000_2	0.10001101_2
1.0001001_2	0.11101110_2	1.0101001_2	0.11000001_2	1.1001001_2	0.10100011_2	1.1101001_2	0.10001100_2
1.0001010_2	0.11101101_2	1.0101010_2	0.11000000_2	1.1001010_2	0.10100010_2	1.1101010_2	0.10001100_2
...	...	...	...	...	...	...	...
1.0010110_2	0.11011010_2	1.0110110_2	0.10110100_2	1.1010110_2	0.10011001_2	1.1110110_2	0.10000101_2
1.0010111_2	0.11011000_2	1.0110111_2	0.10110011_2	1.1010111_2	0.10011000_2	1.1110111_2	0.10000100_2
1.0011000_2	0.11010111_2	1.0111000_2	0.10110010_2	1.1011000_2	0.10010111_2	1.1111000_2	0.10000100_2
1.0011001_2	0.11010101_2	1.0111001_2	0.10110001_2	1.1011001_2	0.10010111_2	1.1111001_2	0.10000011_2
1.0011010_2	0.11010100_2	1.0111010_2	0.10110000_2	1.1011010_2	0.10010110_2	1.1111010_2	0.10000011_2
1.0011011_2	0.11010011_2	1.0111011_2	0.10101111_2	1.1011011_2	0.10010101_2	1.1111011_2	0.10000010_2
1.0011100_2	0.11010001_2	1.0111100_2	0.10101110_2	1.1011100_2	0.10010101_2	1.1111100_2	0.10000010_2
1.0011101_2	0.11010000_2	1.0111101_2	0.10101101_2	1.1011101_2	0.10010100_2	1.1111101_2	0.10000001_2
1.0011110_2	0.11001111_2	1.0111110_2	0.10101100_2	1.1011110_2	0.10010011_2	1.1111110_2	0.10000001_2
1.0011111_2	0.11001101_2	1.0111111_2	0.10101011_2	1.1011111_2	0.10010011_2	1.1111111_2	0.10000000_2

Q. *What's this got to do with Lisp?*

A. Theorems are not proved about artifacts (like billiard balls, artillery shells, moon rockets, or microprocessors); theorems are proved about *mathematical descriptions* or *models* of those artifacts. And Lisp is our language of choice.

The Formal Model of the Code

```
(defun FDIV (p d mode)
 (let*
   ((sd0  (eround (lookup d)             '(exact  17  8)))
    (dr   (eround d                      '(away   17 32)))
    (sdd0 (eround (* sd0 dr)             '(away   17 32)))
    (sd1  (eround (* sd0 (comp sdd0 32)) '(trunc  17 32)))
    (sdd1 (eround (* sd1 dr)             '(away   17 32)))
    (sd2  (eround (* sd1 (comp sdd1 32)) '(trunc  17 32)))
    ...
    (qq2  (eround (+ q2 q3)              '(sticky 17 64)))
    (qq1  (eround (+ qq2 q1)             '(sticky 17 64)))
    (fdiv (round   (+ qq1 q0)            mode)))
   (or (first-error sd0 dr sdd0 sd1 sdd1 ... fdiv)
       fdiv)))
```

The K5 FDIV Theorem (1200 lemmas)

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

(by Moore, Lynch and Kaufmann, in 1995, *before the K5 was fabricated*)

AMD Athlon 1997

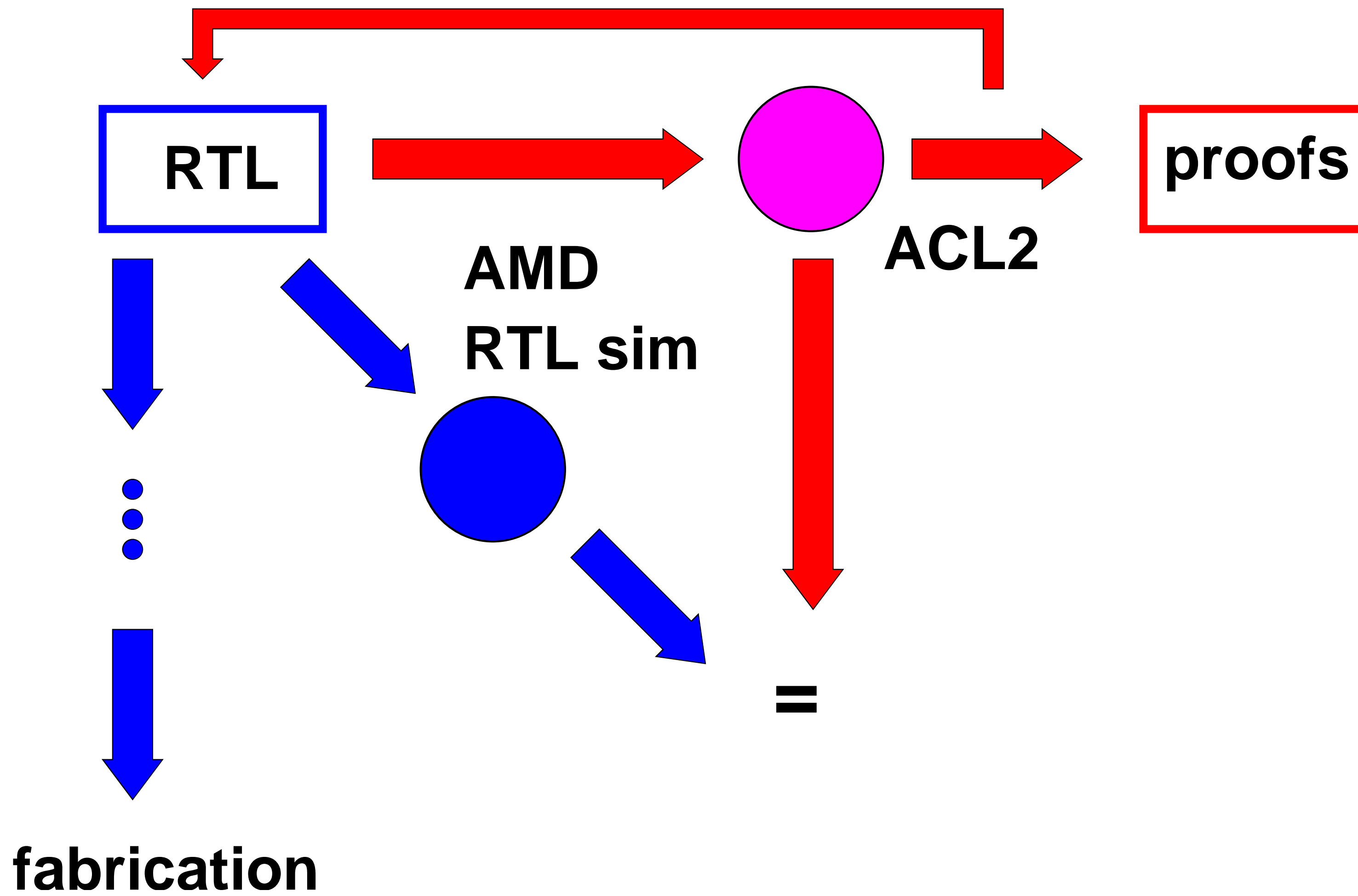
All elementary floating-point operations, `FADD`, `FSUB`, `FMUL`, `FDIV`, and `FSQRT`, on the AMD Athlon were

- specified in ACL2 to be IEEE compliant,
- proved to meet their specifications, and
- the proofs were checked mechanically.

AMD Athlon FMUL

```
module FMUL;  // sanitized from AMD Athlon(TM)
              // by David Russinoff and Art Flatau
//**************************************************
// Declarations
//**************************************************
//Precision and rounding control:
‘define SNG   1’b0      // single precision
‘define DBL   1’b1      // double precision
‘define NRE   2’b00     // round to nearest
‘define NEG   2’b01     // round to minus infinity
‘define POS   2’b10     // round to plus infinity
```

```
//Parameters:
input x[79:0];           //first operand
input y[79:0];           //second operand
input rc[1:0];           //rounding control
input pc;                //precision control
output z[79:0];          //rounded product
//************************************************
// First Cycle
//************************************************
//Operand fields:
sgnx = x[79]; sgny = y[79];
expx[14:0] = x[78:64]; expy[14:0] = y[78:64];
```


RTL
proofs
ACL2
AMD
RTL sim
=
fabrication

The ACL2 proofs uncovered bugs that had remained hidden through hundreds of millions of test cases in RTL simulators.

The bugs were fixed and the new RTL verified *before* the Athlon was fabricated.

This work was done primarily by David Russinoff and Art Flatau, of AMD.

Some Interesting Software Verified

- Motorola CAP DSP (Brock)

  “$\neg hazards(s)$
  $\rightarrow microarch(s) = isa(s)$”

  - more than 100 pages of models

  - established correctness of isa as a simulator

- isa (in ACL2) executes **3 times faster** than Motorola's own $microarch$ simulator (in C) and is proved to be bit- and cycle-accurate!

This is verified software that is worth something!

- FDIV on AMD K5 (Moore, Kaufmann, Lynch)
- fp RTL for AMD Athlon^{TM} processor and AMD Opteron^{TM} processor (Russinoff, Flatau, Kaufmann, Smith, Sumners)
- FDIV and FSQRT on IBM Power 4 (Sawada)

- soundness of the Ivy proof checker (McCune, Shumsky)
- correctness of BDD implementation (60% CUDD speed without reordering) (Sumners)
- Union Switch & Signal post-compiler checker (Bertolli)

- Rockwell instruction set equivalence theorems (Greve, Wilding)
- Rockwell AAMP7 separation kernel in microcode (Greve, Wilding)
- Rockwell / aJile Systems JEM1 (Hardin, Greve, Wilding)

- Sun CLDC JVM model (700 pages) (Liu)
- properties of various Java classes and methods via javac (Moore, Smith)
- correctness of Sun JVM class loader (Liu)
- correctness of Sun JVM bytecode verifier (Liu) (in progress)

- Bryant's Y86 Verilog (microarchitecture correspondence) (Ray)
- Dijkstra's shortest path (Moore, Zhang)
- Unicode reader (Davis)
- See also workshops and seminar links from ACL2 home page

Lisp as a Meta Language

Our proofs are often based on operational semantics.

We define interpreters (for netlists, bytecode, etc.) in Lisp.

We map programs into this code via conventional compilers (gcc, javac).

Via this “deep embedding” we move up and down the abstraction hierarchy while staying in one logic and proof system.

AMD has a sophisticated “shallow embedding” of a design language that incorporates automatic verified simplification.

JVM Operational Semantics

Our “M6” model is based on an implementation of the J2ME KVM. It executes most J2ME Java programs (except those with significant I/O or floating-point).

M6 supports all CLDC data types, multi-threading, dynamic class loading,

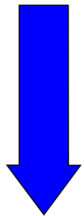
class initialization and synchronization via monitors.

We have translated the entire Sun CLDC API library implementation into our representation with 672 methods in 87 classes. We provide implementations for 21 out of 41 native APIs that appear in Sun's CLDC API library.

We prove theorems about bytecoded methods with the ACL2 theorem prover.

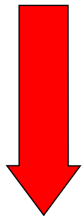
This work is supported by a gift from Sun Microsystems.

.java



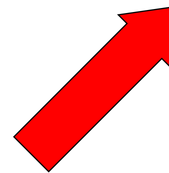
javac

.class



jvm2acl2

.lisp



Theorems

“‘fact(5)=120’”

“‘fact(n)=n!’”

ACL2 Demo 2

Why are we succeeding?

Reason 1: Our modeling language and logic is Lisp:

- expressive
- flexible
- rugged

- widely supported on many platforms
- extremely efficient
- supported with excellent editors and debugging tools provided by others

Reason 2: We have invested 34 years

- striving for perfection
- supporting efficient execution
- integrating a wide variety of proof techniques
- engineering for industrial scale formulas

- developing reusable “books” (lemma collections)

- investing effort in documentation (books, online text, HTML, Emacs info)

- active email lists and user support

Reason 3: We have chosen the right problems. In our applications, the models typically

- describe actual digital artifacts
- are easily related to the artifacts
- are executed to convince designers and management that verification is relevant

- are useful as simulation engines
- permit mathematical abstraction and proof of important properties

Drawbacks

The ACL2 user must be trained to *use* the tool, not *fight* it.

The language is not as expressive as many would like (but it is executable).

Finding appropriate lemmas is often a hard (mathematical) problem.

Developing compatible collections of theorems takes system design (not just mathematical) talent.

Our Hypothesis

The “high cost” of formal methods

– to the extent the cost is high –

is a *historical anomaly* due to the fact that virtually every project formally recapitulates the past.

The use of mechanized formal methods will ultimately

- *decrease* time-to-market, and
- *increase* reliability.

Conclusion

Mechanical reasoning systems are changing the way complex digital artifacts are built.

Complexity not an argument *against* formal methods.

It is an argument *for* formal methods.