The ACL2 Theorem Prover and How It Came to be Used in Industry

(A Sprint through 45 Years of Verification History)

J Strother Moore
Department of Computer Science
University of Texas at Austin

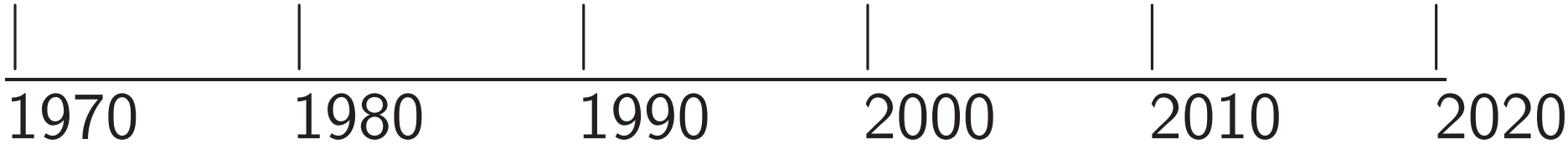
Boyer-Moore-Kaufmann Project

Edinburgh Pure Lisp Theorem Prover [*BM, 1973*]

A Computational Logic [*BM, 1979*]

NQTHM [*BM, 1981*]

ACL2 [*BM,KM 1989–present*]



Vision: To build a practical verification system: axiomatize a functional programming language, build a theorem prover for it, and use them to model and verify computational artifacts.

ACL2

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

A fully integrated verification environment for a practical applicative subset of an ANSI standard programming language

{kaufmann,moore}@cs.utexas.edu

http://www.cs.utexas.edu/users/moore/acl2

I can't explain ACL2 in the time we have, I can only highlight some capabilities.

History of Theorems Proved (by Our Provers)

simple list processing

academic math and CS

breakthrough commercial applications

routine industrial use

A Few Axioms

- $\texttt{t} \neq \texttt{nil}$
- $\texttt{x} = \texttt{nil} \rightarrow (\texttt{if x y z}) = \texttt{z}$
- $\texttt{x} \neq \texttt{nil} \rightarrow (\texttt{if x y z}) = \texttt{y}$
- $(\texttt{car (cons x y)}) = \texttt{x}$
- $(\texttt{cdr (cons x y)}) = \texttt{y}$
- $(\texttt{endp nil}) = \texttt{t}$
- $(\texttt{endp (cons x y)}) = \texttt{nil}$

A Common Abbreviation

```
'(1 2 3)
```

=

```
(cons 1 '(2 3))
```

=

...

```
(cons 1 (cons 2 (cons 3 nil)))
```

Definition – List Concatenation

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

Definition – List Concatenation

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

Thm: (ap nil y) $=$ y

Thm: (ap (cons u v) y) $=$ (cons u (ap v y))

Definition – List Concatenation

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

Thm: (ap nil y) = y

Thm: (ap (cons u v) y) = (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
(ap '(1 2 3) '(4 5 6))
```

Definition – List Concatenation

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

Thm: (ap nil y) = y

Thm: (ap (cons u v) y) = (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
(ap (cons 1 (cons 2 (cons 3 nil))) '(4 5 6))
```

Definition – List Concatenation

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

Thm: (ap nil y) = y

Thm: (ap (cons u v) y) = (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
(cons 1 (ap (cons 2 (cons 3 nil)) '(4 5 6)))
```

Definition – List Concatenation

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

Thm: (ap nil y) = y

Thm: (ap (cons u v) y) = (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
(cons 1 (cons 2 (ap (cons 3 nil) '(4 5 6))))
```

Definition – List Concatenation

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

Thm: (ap nil y) = y

Thm: (ap (cons u v) y) = (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
(cons 1 (cons 2 (cons 3 (ap nil '(4 5 6)))))
```

Definition – List Concatenation

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

Thm: (ap nil y) = y

Thm: (ap (cons u v) y) = (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
(cons 1 (cons 2 (cons 3 '(4 5 6))))
```

Definition – List Concatenation

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

Thm: (ap nil y) $=$ y

Thm: (ap (cons u v) y) $=$ (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
(cons 1 (cons 2 '(3 4 5 6)))
```

Definition – List Concatenation

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

Thm: (ap nil y) = y

Thm: (ap (cons u v) y) = (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
(cons 1 '(2 3 4 5 6))
```

Definition – List Concatenation

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

Thm: (ap nil y) = y

Thm: (ap (cons u v) y) = (cons u (ap v y))

```
(ap '(1 2 3) '(4 5 6)) =
'(1 2 3 4 5 6)
```

Theorem ap is associative (1971)

$\forall a \forall b \forall c$: ap(ap(a,b),c) = ap(a,ap(b,c)).

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

Proof: by induction on a.

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (ap b c)
       (ap a (ap b c)))
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (ap b c)
       (ap a (ap b c)))
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (ap b c)
       (ap b c))
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (ap b c)
       (ap b c))
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))

Proof: by induction on a.

Base Case:  (endp a).
T
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))

Proof: by induction on a.

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

         (ap (cdr a) (ap b c)))
```

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (ap (ap (cdr a) b) c)        ; Ind Hyp
       (ap (cdr a) (ap b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))

Proof: by induction on a.

Induction Step: (not (endp a)).
T
```

```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

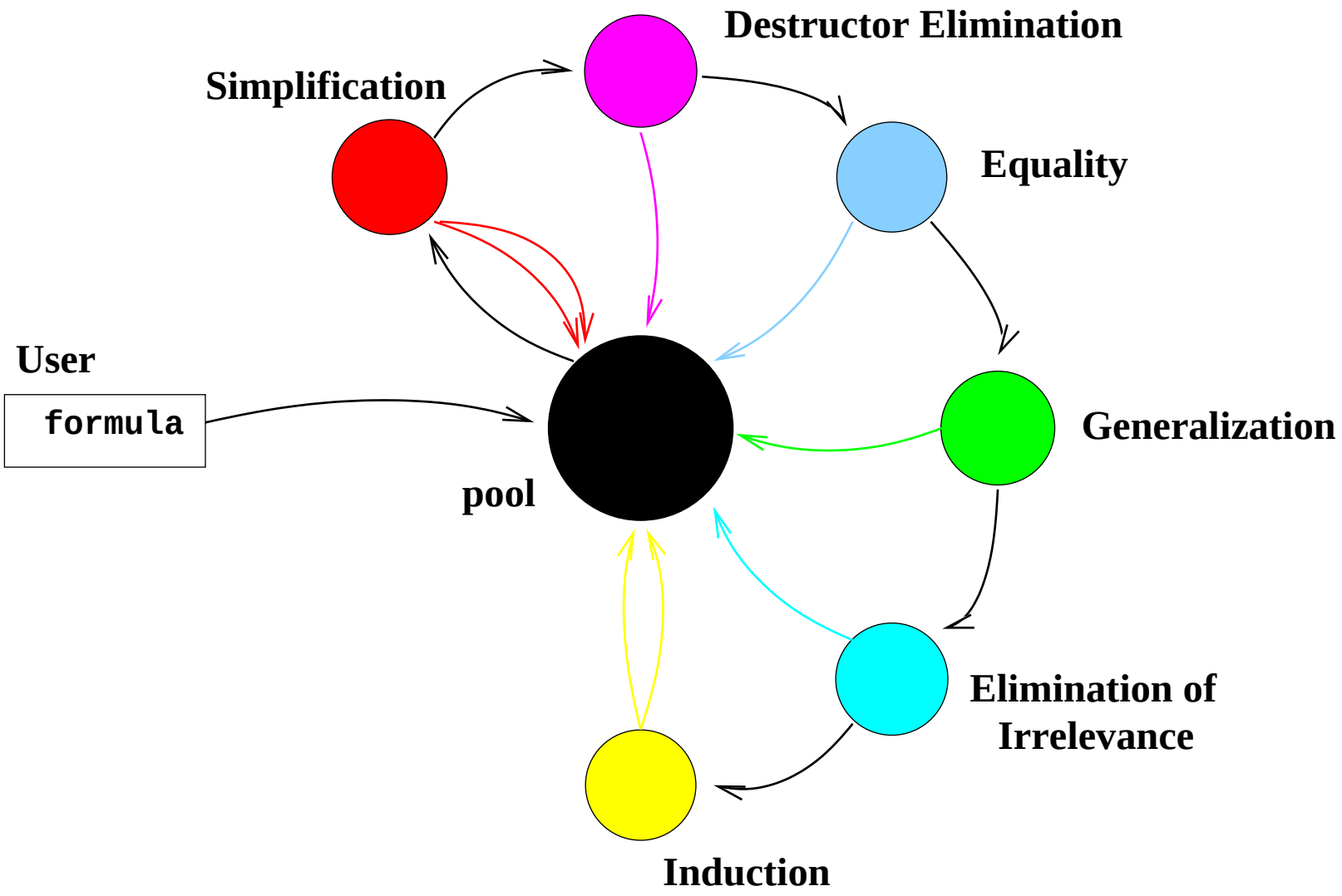
Proof: by induction on a.

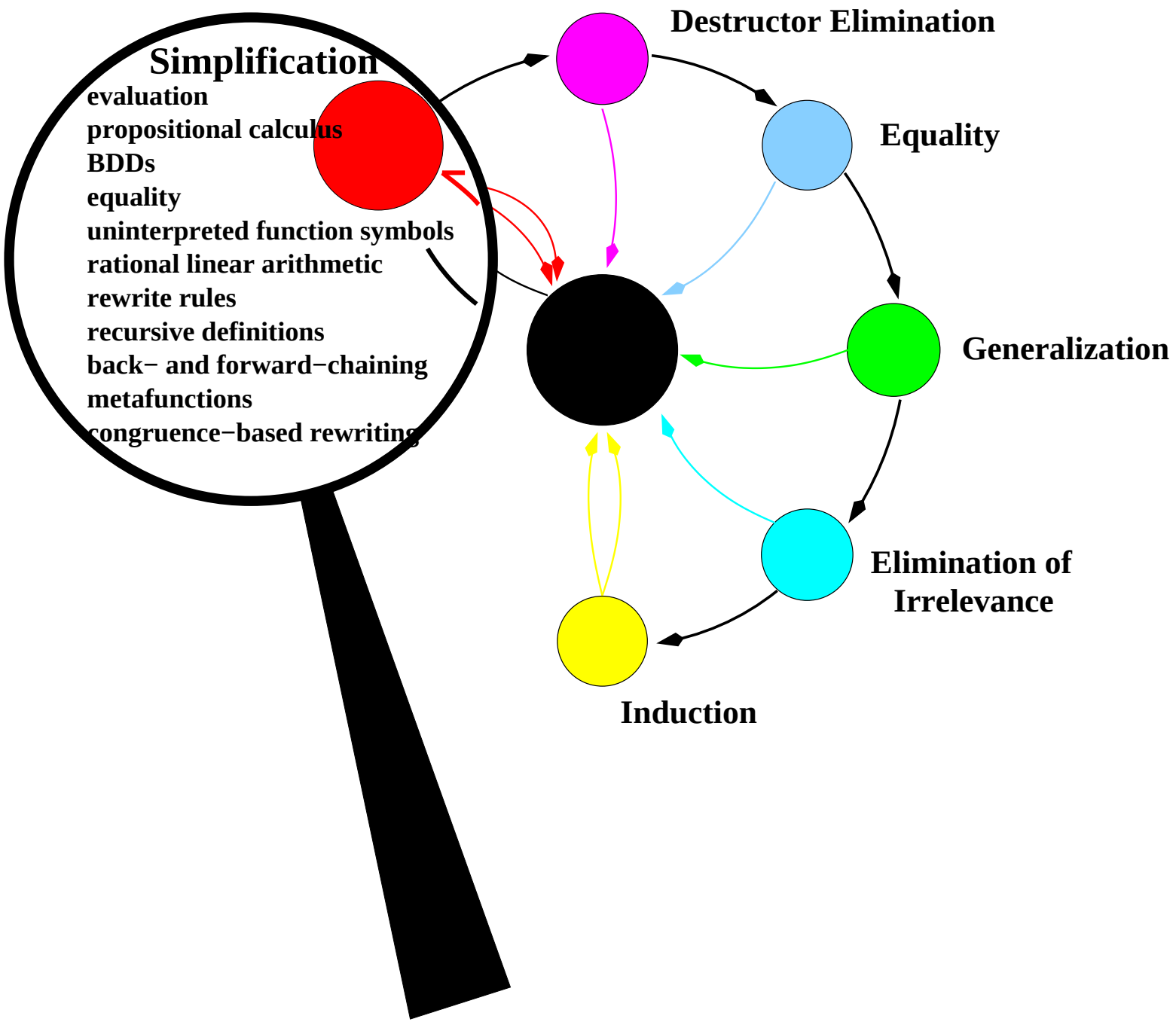
Q.E.D.

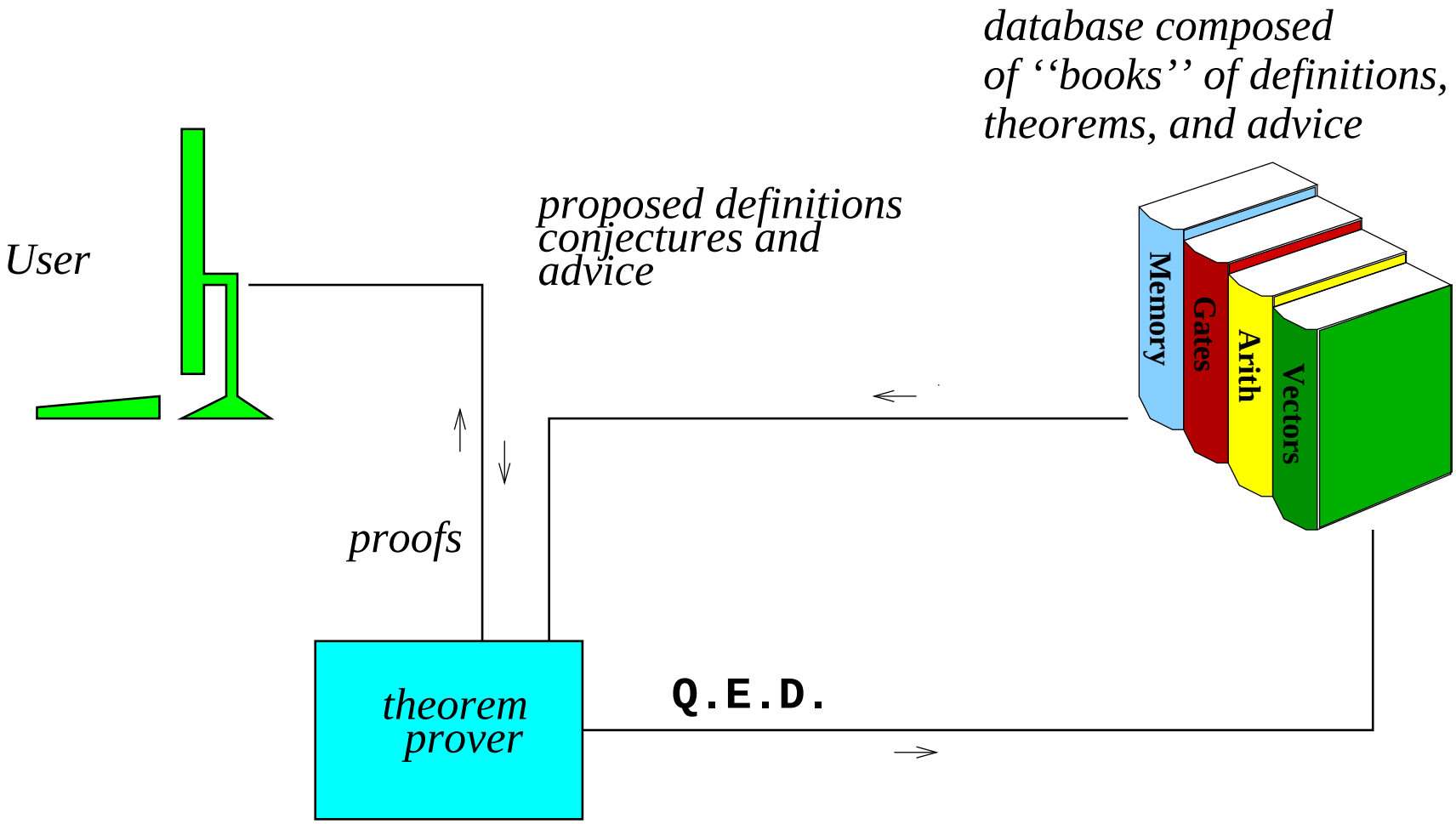
ACL2 Demo 1

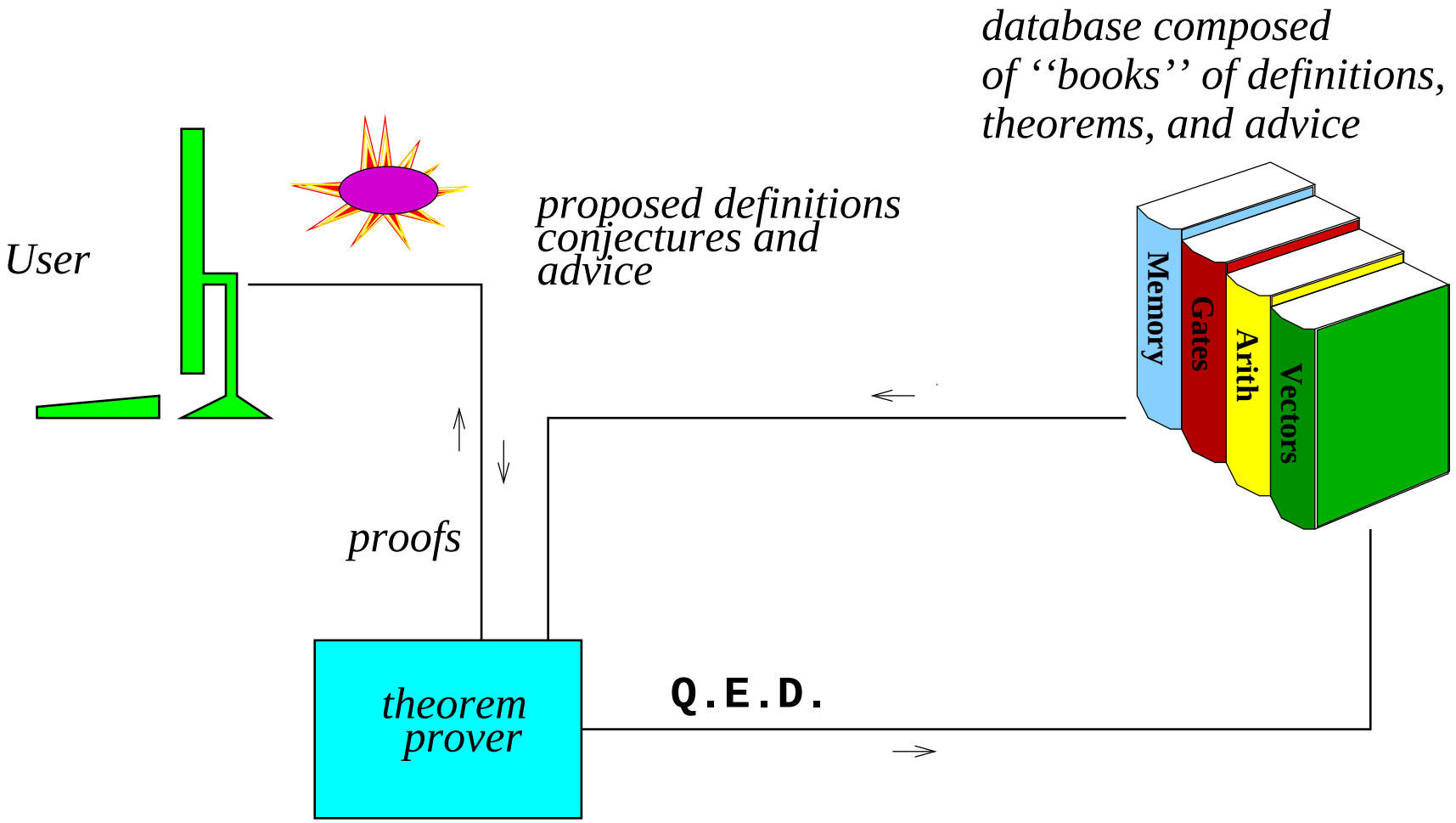
Notes: The four demos in this talk are with ACL2 even though many of the proofs in the first few demos were first done with earlier Boyer-Moore provers.

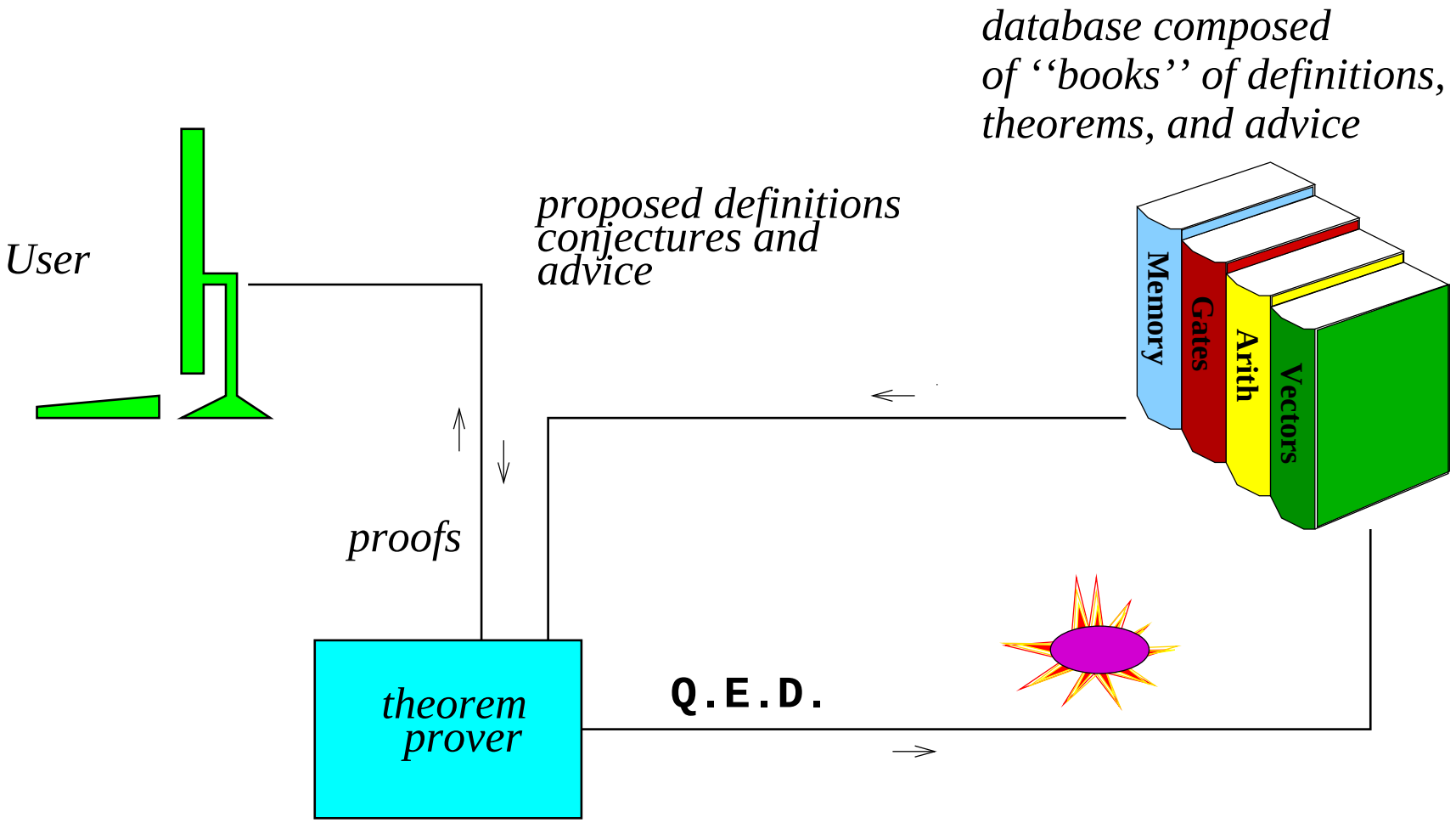
The ACL2 image used here has been configured to support these four demos.

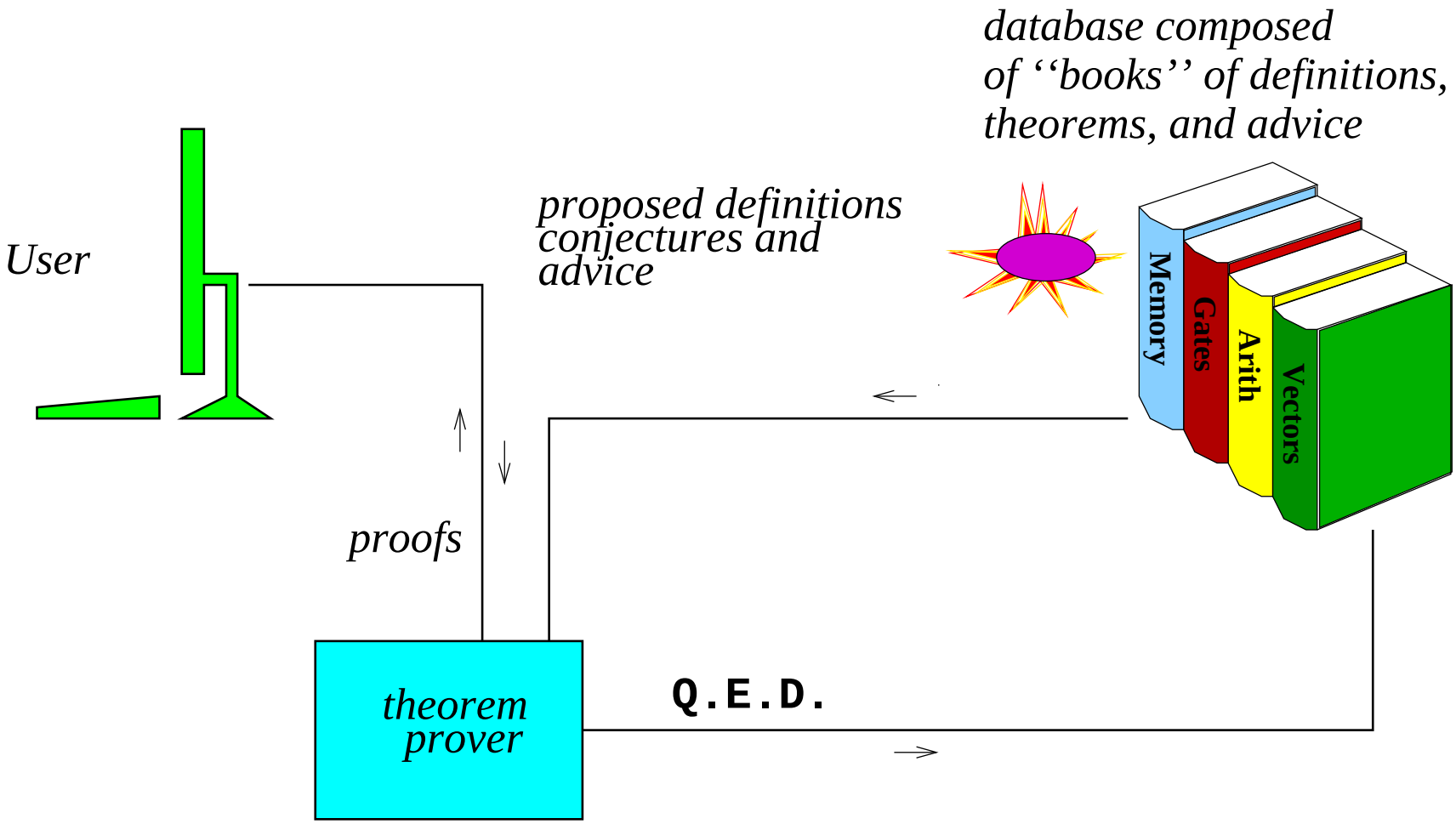

Destructor Elimination
Simplification
Equality
User
formula
Generalization
pool
Elimination of
Irrelevance
Induction


Simplification
evaluation
propositional calculus
BDDs
equality
uninterpreted function symbols
rational linear arithmetic
rewrite rules
recursive definitions
back− and forward−chaining
metafunctions
congruence−based rewriting
Destructor Elimination
Equality
Generalization
Elimination of
Irrelevance
Induction


database composed
of ‘‘books’’ of definitions,
theorems, and advice
User
proposed definitions
conjectures and
advice
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.


database composed
of ‘‘books’’ of definitions,
theorems, and advice
User
proposed definitions
conjectures and
advice
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.


database composed
of ‘‘books’’ of definitions,
theorems, and advice
User
proposed definitions
conjectures and
advice
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.


database composed
of ‘‘books’’ of definitions,
theorems, and advice
User
proposed definitions
conjectures and
advice
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.

ACL2 Community Books (2017)

`https://github.com/acl2/acl2/books/`
contains ∼6,000 user-supplied books.

ACL2 Demo 2

History of Theorems Proved

simple list processing

academic math and CS

breakthrough commercial applications

routine industrial use

Academic Math (Nqthm, 1980s)

- undecidability of the halting problem
  (18 lemmas)
- invertibility of RSA encryption
  (172 lemmas)
- Gauss' law of quadratic reciprocity [Russinoff]
  (348 lemmas)
- Gödel's First Incompleteness Theorem [Shankar]
  (1741 lemmas)

Academic CS (Nqthm, 1980s)

- The CLI Verified Stack:
  - microprocessor: gates to machine code [Hunt]
  - assembler-linker-loader (3326 lemmas) [Moore]
  - compilers [Young, Flatau]
  - operating system [Bevier]
  - applications [Wilding]

These theorems guarantee that a property proved about an app holds when it is compiled, assembled, linked, loaded, and run on the gate-level machine.

ACL2 Demo 3

Seeing Nqthm struggle with "large" models like Piton and other components of the CLI verified stack convinced us to re-implement it with scaling in mind.

Thus was born ACL2 (1989).

Hisory of Theorems Proved

simple list processing

academic math and CS

breakthrough commercial applications

routine industrial use

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]`
...	...	= ...	...
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
      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:

$0.083 \times 430.0000 = 35.6900000 \approx 36.000000 = q_0$

$0.083 \times \text{-}2.0000 = \text{-}.1660000 \approx \text{-}.170000 = q_1$

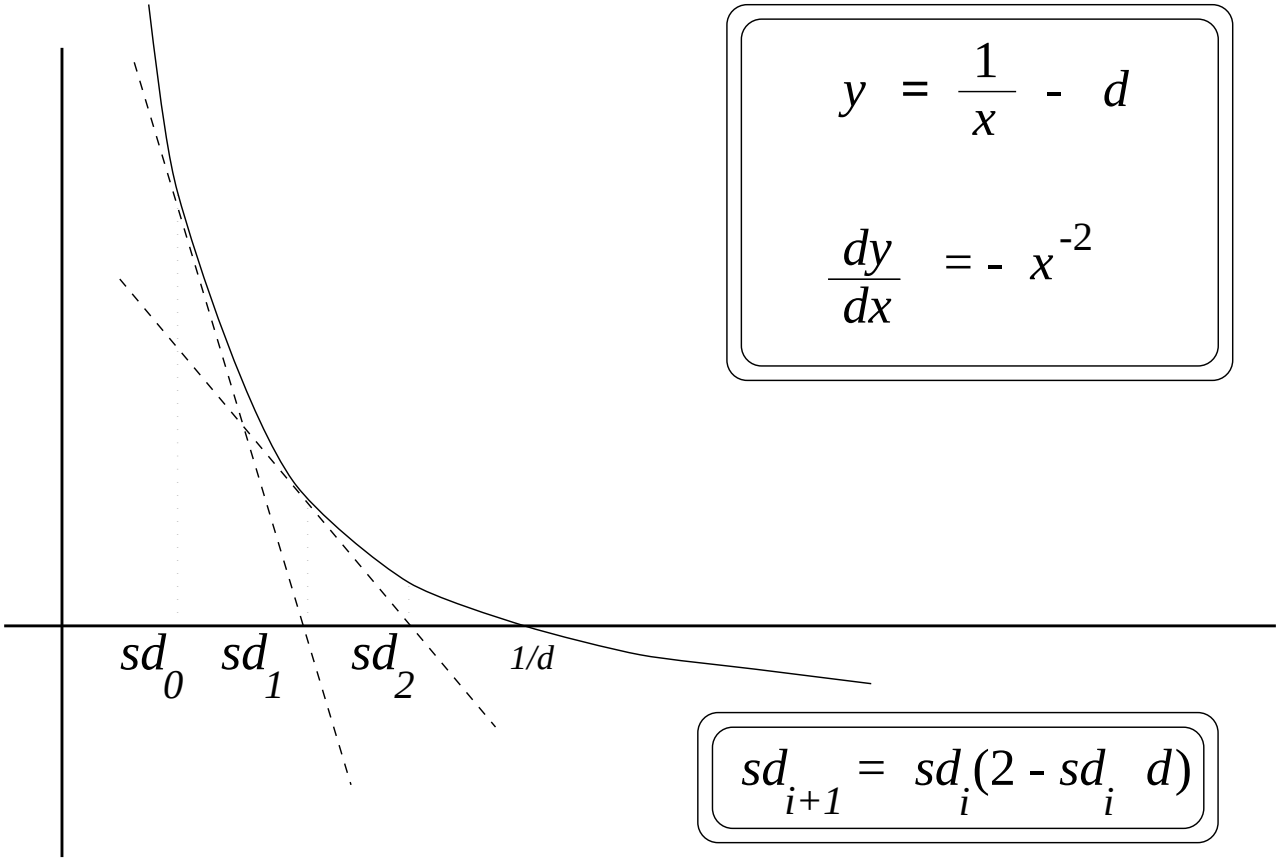
$0.083 \times .0400 = .0033200 \approx .003400 = q_2$

$0.083 \times \text{-}.0008 = \text{-}.0000664 \approx \text{-}.000067 = q_3$

Summation of Quotient Digits:

$q_0 + q_1 + q_2 + q_3 = 35.833333$

Computing the Reciprocal



$$y = \frac{1}{x} - d$$

$$\frac{dy}{dx} = -x^{-2}$$

$$sd_{i+1} = sd_i(2 - sd_i\ d)$$

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

The Futility of Testing

If AMD builds this, will it work?

A bug in this design could cost AMD hundreds of millions of dollars.

On Sunway TaihuLight (93 petaflops $= 93 \times 2^{50}$ operations per second), testing all possible cases would take

2,726,112,523,746,722,547,161,199

$\sim 2.7 \times 10^{24}$ *years*.

The library of floating point lemmas and the main theorem were proved with ACL2 under the direction of two ACL2 users and the designer of the FDIV algorithm.

The proofs took 9 weeks starting from Peano's axioms.

The proofs were completed before the K5 was fabricated.

9 weeks $<$ 2,726,112,523,746,722,547,161,199 years

The library was used in subsequent proofs.

By 1997, AMD had

- built software to translate their in-house hardware design language to ACL2
- used the tool to generate ACL2 functions modeling all the elementary floating point arithmetic on the soon-to-be fabricated AMD Athlon microprocessor

- tested the ACL2 functions by running them on AMD's standard floating-point test suite ($>$ 100 million arithmetic problems) and compared the answers to AMD's design simulator

- proved the ACL2 functions compliant with the IEEE Standard

- found (and fixed) 3 design errors not exposed by the 100 million tests

Other Early Industrial Users of ACL2

- Motorola: DSP and microcode proofs
- AMD: floating-point on Opteron
- Rockwell-Collins: silicon JVM chip, AAMP7 crypo-box, Greenhills OS
- IBM: Power 4 FDIV and SQRT
- Sun Microsystems (via contract): Sun JVM class loader and byte-code verifier

ACL2 Demo 4

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

  public static void main(String[] args){
      int n = Integer.parseInt(args[0], 10);
      System.out.println(fact(n));
      return;
    }
}
```

History of Theorems Proved

simple list processing

academic math and CS

breakthrough commercial applications

routine industrial use

In 2007, Centaur Technology, Inc., challenged the ACL2 community to verify its floating-point adder:

- handles single (32-bit), double (64-bit) and extended (80-bit) additions
- pipelined to deliver 4-results per cycle
- 33,700 lines in 680 Verilog modules
- 1074 input signals (including 26 clocks) and 374 output signals

Done! After exposing and fixing one very rare bug.

(The bug occurred on exactly one pair of 80-bit inputs, i.e., 1 case of 2^{160} cases.)

ACL2 at Centaur Today

ACL2 is an indispensable part of the Centaur design process

Centaur FV team consists of 3 full-time employees and a couple of interns

Centaur has an ACL2 specification of the x86

Validated by routinely running millions of tests comparing ACL2 x86 to Intel, AMD, and Centaur hardware

The ACL2 tool-chain translates the entire Centaur design (700,000+ lines of Verilog) into a formal object in a few minutes

The translated model is validated by running millions of tests against Cadence NC Verilog and Synopsys VCS Verilog simulators

All functional-correctness proofs are re-run nightly (on a cluster of 154 CPUs with a total of 2TB RAM)

"Bugs introduced today are found tonight and fixed tomorrow."

Highlighted Strengths of ACL2

- adequate logical expressivity (to capture design and specs)
- adequate capacity (to manipulate multi-MB formulas)
- efficient execution (to do 100s of millions of sim runs)
- automatic proof discovery after typical design changes (to enable nightly runs)

Advantages for Centaur

- high confidence in design correctness (enabling "riskier" design changes)
- high confidence in inter-generational compatibility
- higher re-use of specifications and modules
- reduced reliance on testing
- reduced time-to-market

x86 ISA in ACL2 (Hunt and Goel)

Supports user- and system-level specifications for the x86 and may serve both to verifying (user- or system-) binary machine code and as a "build-to" spec for designers.

Performance:

user level: $\sim$ 3.3 million ips

system level: $\sim$ 912,000 ips

Other Ongoing Industrial Projects

- AMD (transaction protocols)
- Intel (elliptic curve crypto)
- Kestrel Institute (Android apps)
- Oracle (floating point)
- Rockwell-Collins (LLVN)
- ARM (floating point)

Industrial Wish List

- more automation (esp in lemma/defn discovery)
- faster execution speed of models in the logic
- better ways to view large formulas
- scripting capabilities
- ability to build GUIs

Things Our Industrial Users Haven't Asked For

- quantifiers
- higher-order functions
- strong typing

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 for the past 20 years has had to formalize (parts of) centuries of mathematics and decades of chip design "shop-lore"

The use of mechanized formal methods

- *decreases* time-to-market,
- *increases* reliability.

Conclusion

Mechanical reasoning systems have changed the way complex digital artifacts are built.

Complexity not an argument *against* formal methods.

It is an argument *for* formal methods.

References

Computer-Aided Reasoning: An Approach, Kaufmann, Manolios, Moore, Kluwer Academic Publishers, 2000.

Computer-Aided Reasoning: ACL2 Case Studies, Kaufmann, Manolios, Moore (eds.), Kluwer Academic Publishers, 2000.

http://www.cs.utexas.edu/users/moore/acl2

Extra Slides

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)

"If p and d are $64, , 15^{+}$ floating point numbers, $d \neq 0$, and $mode$ is an IEEE rounding mode, then $\texttt{FDIV}(p, d, mode) = \mathrm{round}(p/d, mode)$."

```
(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))))
```

[Moore, Kaufmann, Lynch – 1995]

A Few of the 1200 AMD Lemmas

Trunc Trunc: If $i \leq j$, then
$\mathrm{trunc}(\mathrm{trunc}(x, i), j) = \mathrm{trunc}(x, i)$.

Sticky Enough: If $mode$ is an IEEE rounding mode with size $n < i$, then
$\mathrm{round}(\mathrm{sticky}(x, i + 2), mode) = \mathrm{round}(x, mode)$.

Sticky Plus: Let $x \neq 0$, such that $\mathrm{trunc}(x, n) = x$ and $1 + e(y) < e(x)$, and $n + e(y) - e(x) < k$.
Then $\mathrm{sticky}(x + y, n) = \mathrm{sticky}(x + \mathrm{sticky}(y, k), n)$.

(Some standard hypotheses have been omitted for brevity.)

ACL2 at Centaur (cont')

Centaur uses ACL2 to build *verified* custom tools for its Verilog designers

Such tools can used by ACL2 because of its *metafunction* (*reflection*) capability

Mechanized formal reasoning and theorem proving are taken for granted

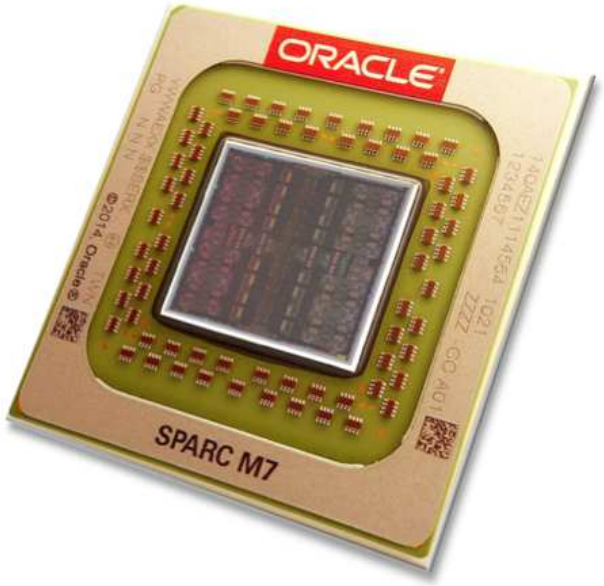
Centaur's Verilog tool-chain is distributed with ACL2 and is used by Intel and Oracle

Reports on Oracle Usage

Verifying Oracle's SPARC processors with ACL2

Greg Grohoski
VP, Hardware Development
Oracle Microelectronics

May 23, 2017



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Summer 2014

- Wrote 1st draft of ACL2 spec for
  - IEEE 754 Standard on Floating-Point Arithmetic
  - Integer division variants
- Verified
  - Floating-point implementations wrt significand
  - Integer division implementations
- Found improvements for SPARC core
  - Reduced look-up tables by 60%
  - Improvements based on careful error analysis
  - Simplified square-root implementation

ORACLE

Copyright © 2017 Oracle and/or its affiliates. All rights reserved. |

Summer 2015

- Validated ACL2 spec of IEEE 754 Standard against 8M vectors
- Verified all 4 floating-point implementations wrt
  - Sign, exponent, and significand
  - All exceptions
- Found more improvements for future SPARC core
  - Reduced latency further for
    - fdivs, fdivd
    - fsqrts, fsqrtd
    - idiv
- Improvements were all proven correct first and then implemented
  - No extensive 24x7 testing needed

ORACLE

Copyright © 2017 Oracle and/or its affiliates. All rights reserved. |

Summer 2016

- Applied ACL2 to other areas
  - Temporal properties
  - Proved absence of starvation for certain instructions
- Looked at verification of complex crypto instructions
  - Verified Montgomery algorithms implementations wrt their math specification
  - Saved 25% latency in one instruction
  - Generated test vectors to exercise special cases for these instructions
- Verified and improved fault-tolerant cache-coherency protocol (like CCIX)
  - "Bake off" between ACL2 (TP) and Murphi/PReach (MC)

ORACLE

Copyright © 2017 Oracle and/or its affiliates. All rights reserved. |

Summer 2017

- Found more improvements for integer division for future SPARC core
  - Further, significant latency reduction for integer division
  - Again, optimization was first verified, then implemented
- Verified another function using Cellular Automata Shift Registers
- Applied ACL2 to prove absence of starvation for certain instructions
  - Another “bake off” between ACL2 (TP) and SVA model checking

ORACLE

Copyright © 2017 Oracle and/or its affiliates. All rights reserved. |

Rockwell Collins Usage

Rockwell Collins

Our Uses of ACL2 to Date

- Microcode Modeling and Proofs
- AAMP7 Information Flow Proofs (GWV Theorem)
  - NSA MILS Accreditation
- Green Hills Information Flow Proofs (GWVr2 Theorem)
  - EAL6+ Accreditation
- AAMP7 Instruction Set Modeling and Proofs
  - Interface to Eclipse-based Debugger
- MicroCryptol Runtime
- Proofs for Guard Prototype (AAMP7 code, vFAAT)
- Data Flow Logic (DFL) for C code
- LLVM Modeling and Proofs
- Other things we can't talk about…

Themes:

- ***Automated High-Level Property Verification for Low-Level Artifacts***
- ***Validation Enabled by Executable Formal Models***

© Copyright 2015 Rockwell Collins, Inc.
All rights reserved.

2

ACL2 Support for Industrial Projects

There have been over 1000 changes to ACL2 since Centaur started using ACL2 in May, 2009. Of those, these were requested by Centaur:

Changes to Existing Features	95
New Features	44
Heuristic and Efficiency Improvements	22
Bug Fixes	72
Changes at the System Level	18
Total due to Centaur	251

Why ACL2 is Successful in Industry

- that was the goal of the project
- efficient, executable logic/programming language with native verifier
- dual-use bit- and cycle-accurate models
- access to Common Lisp programming (via trust tags)
- automatic prover with "a human in the loop"

- encourages development of domain-specific automatic provers allowing *proof maintenance* as designs evolve

- rugged, well documented, free, open source form, many useful books, and a fairly unrestrictive license

- coherent user community devoted to making mechanized verification practical

- industry needs help: their designs are too complicated to get right without mechanized reasoning

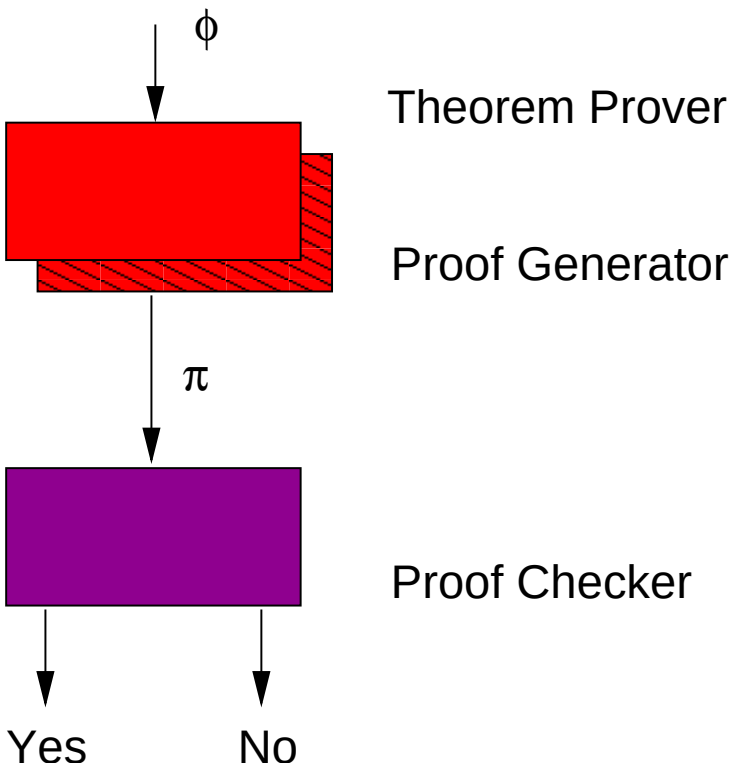
How Do We Know ACL2 is Sound?

"Trust us!" – *Kaufmann and Moore*

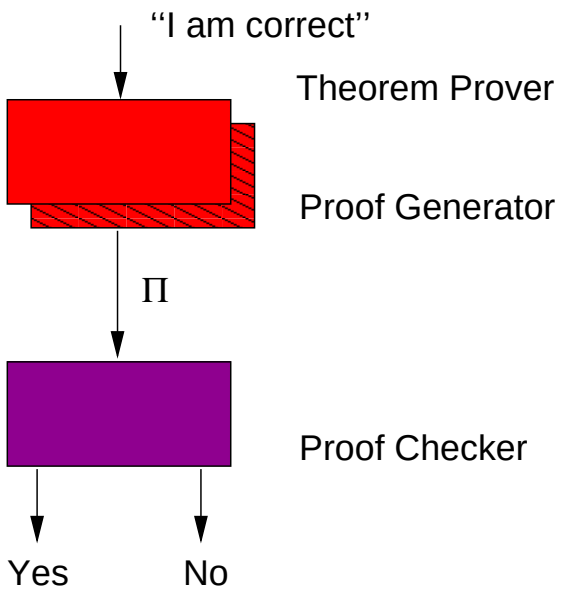
Obviously, we would like to prove it correct.

But with what prover?

Meaning of Correctness

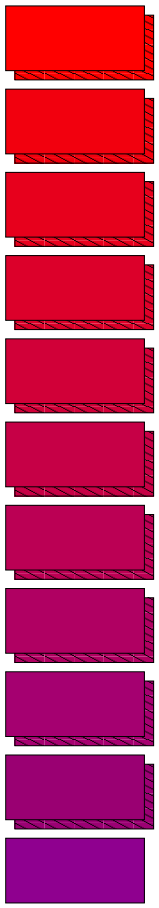


Plan



- Prove "I am correct" with Theorem Prover
- Generate *that* proof Π
- Check Π with Proof Checker
- Never generate another low-level proof

Jared Davis' Stack "Milawa"



Level

11 Induction and other tactics

10 Conditional rewriting

9 Evaluation and unconditional rewriting

8 Audit trails (in prep for rewriting)

7 Case splitting

6 Factoring, splitting help

5 Assumptions and clauses

4 Miscellaneous ground work

3 Rules about primitive functions

2 Propositional reasoning

1 Primitive proof checker