Machines Reasoning about Machines

A Personal Perspective

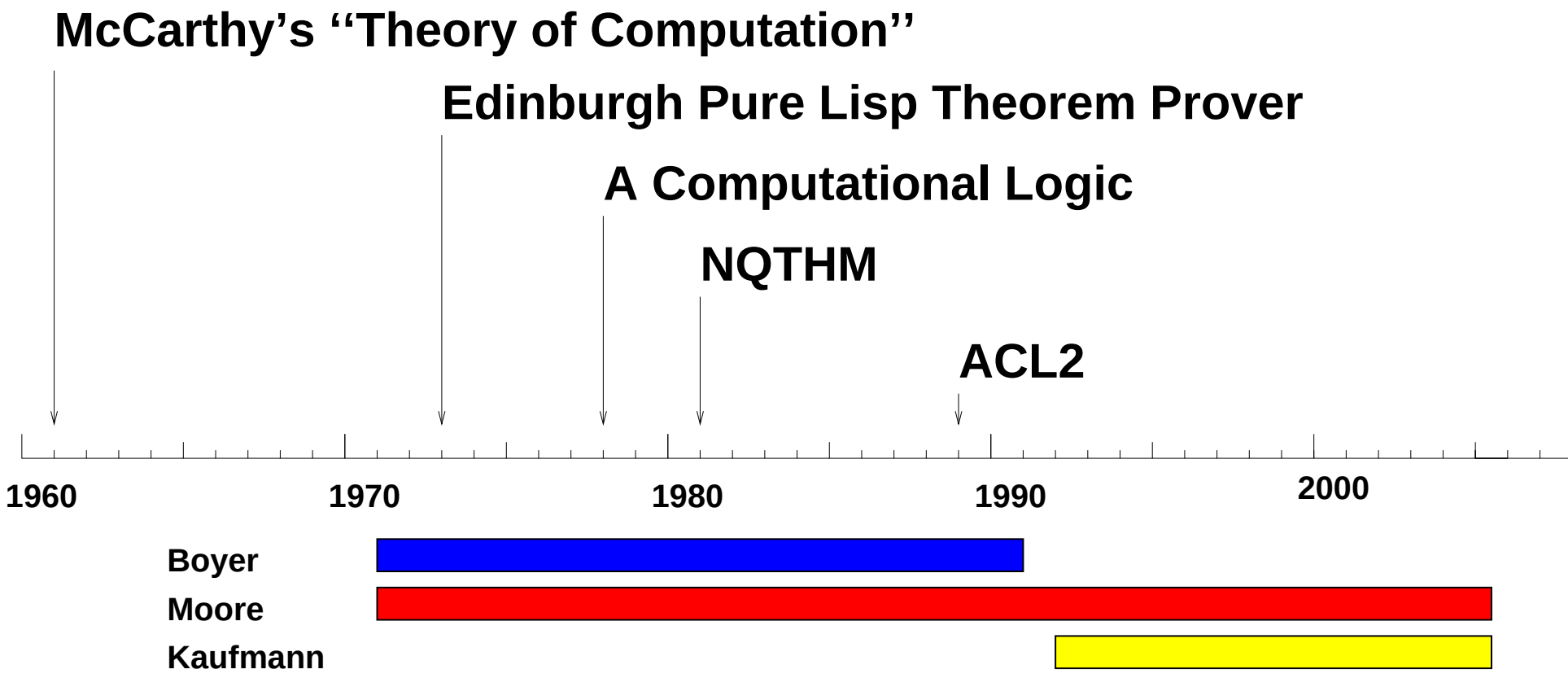
J Strother Moore
Department of Computer Sciences
University of Texas at Austin

Is McCarthy's Dream Practical?

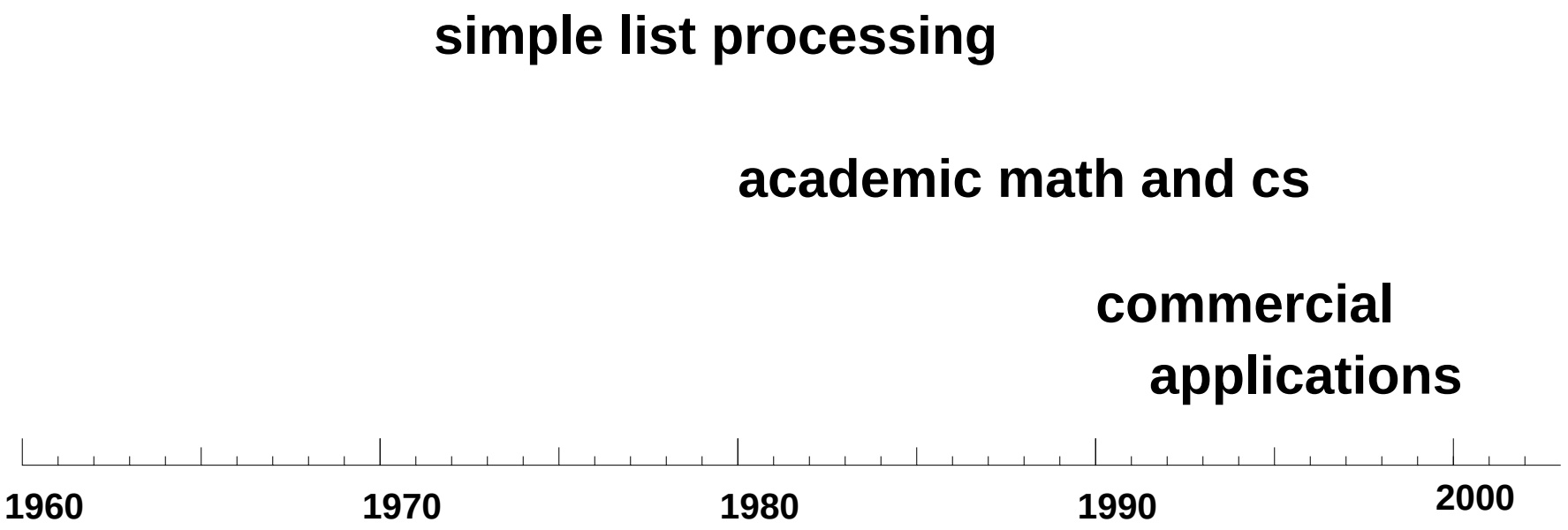
Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program.

— *John McCarthy, "A Basis for a Mathematical Theory of Computation," 1961*

Boyer-Moore Project



Theorems Proved



Topics We'll Touch Upon

Progressively more challenging applications.

How theorem provers work.

What can be done with them today.

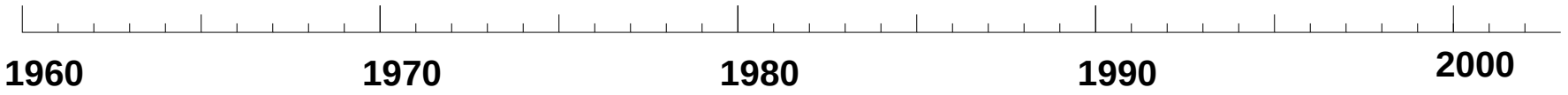
Various ways to approach modeling and code proofs.

Theorems Proved

simple list processing

academic math and cs

commercial applications



Theorems Proved: 1970s

- append is associative:

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

$\forall a \forall b \forall c$

append(append(a,b),c)

$=$

append(a,append(b,c)).

A Few Axioms

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

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

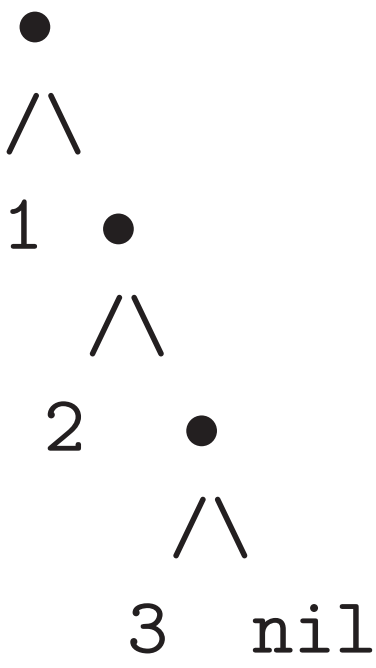
$$\texttt{x} \neq \texttt{nil} \rightarrow (\texttt{if x y z}) = \texttt{y}$$

$$(\texttt{car } (\texttt{cons x y})) = \texttt{x}$$

$$(\texttt{cdr } (\texttt{cons x y})) = \texttt{y}$$

```
(endp nil) = t

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

Ordered Pairs as Lists



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

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

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

```
(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)))
```

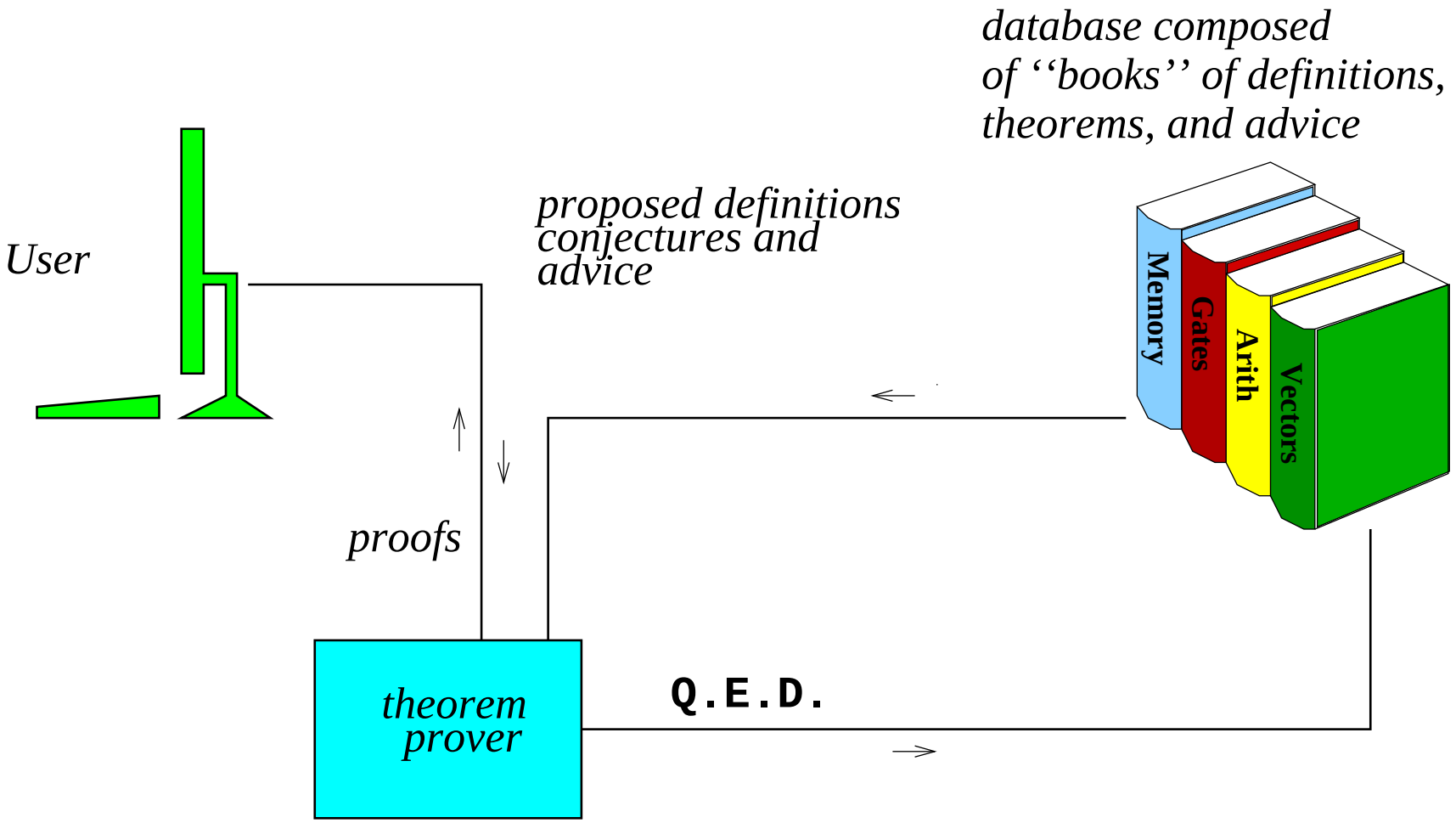
```
(equal (append (append a b) c)
       (append a (append b c)))

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.

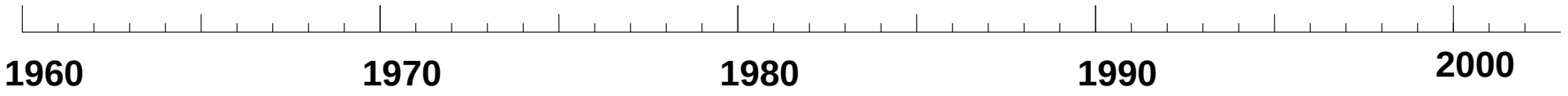

database composed
of ‘‘books’’ of definitions,
theorems, and advice
User
proposed definitions
conjectures and
advice
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.

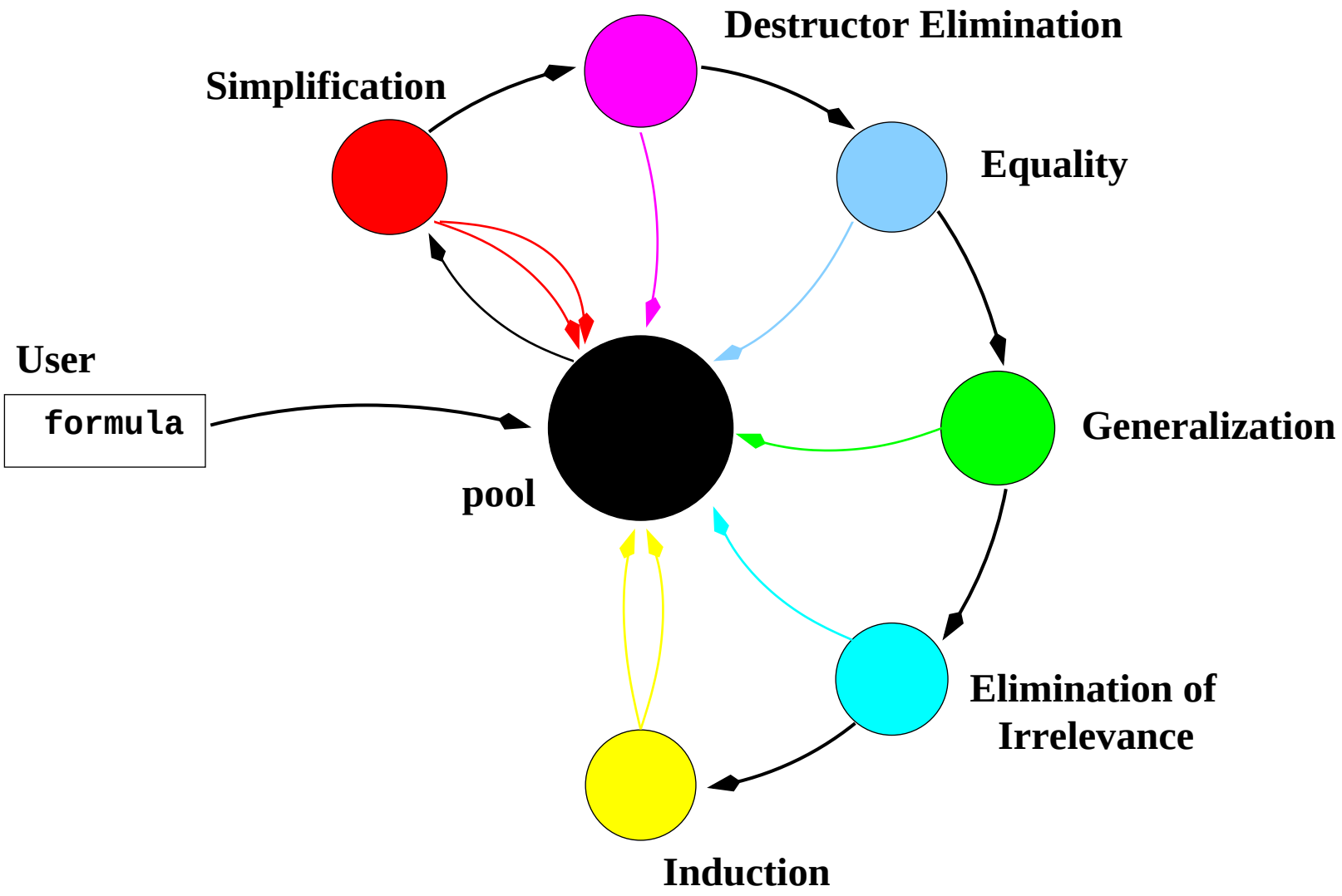
ACL2 Demo 1

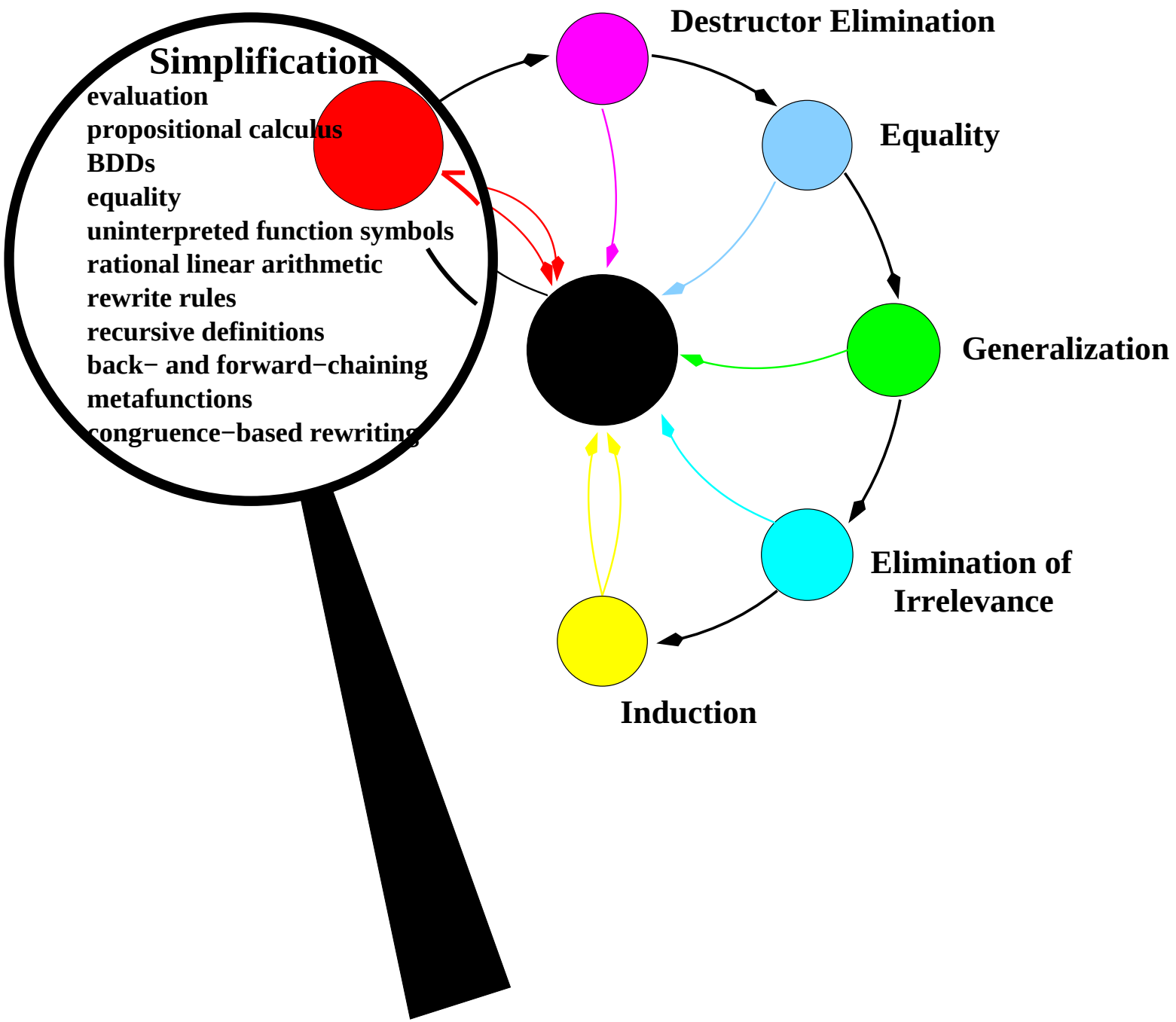
simple list processing

academic math and cs

commercial applications




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

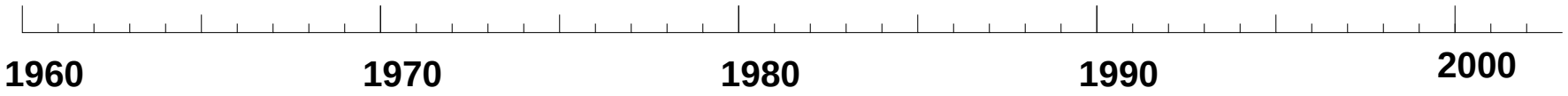

Simplification
evaluation
propositional calculus
BDDs
equality
uninterpreted function symbols
rational linear arithmetic
rewrite rules
recursive definitions
back− and forward−chaining
metafunctions
congruence−based rewriting
Destructor Elimination
Equality
Generalization
Elimination of
Irrelevance
Induction

Theorems Proved

simple list processing

academic math and cs

commercial applications



1980s Academic Math

- 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)

1980s Academic CS

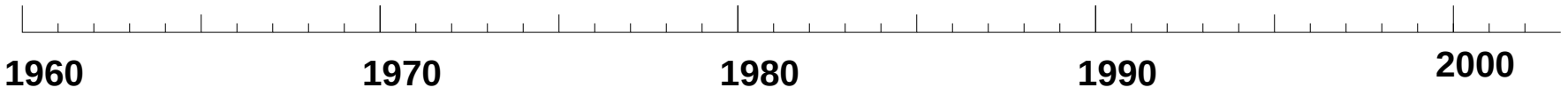
- The CLInc Verified Stack:
  - microprocessor: gates to machine code [Hunt]
  - assembler-linker-loader (3326 lemmas)
  - compilers [Young, Flatau]
  - operating system [Bevier]

Theorems Proved

simple list processing

academic math and cs

commercial applications



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:

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

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

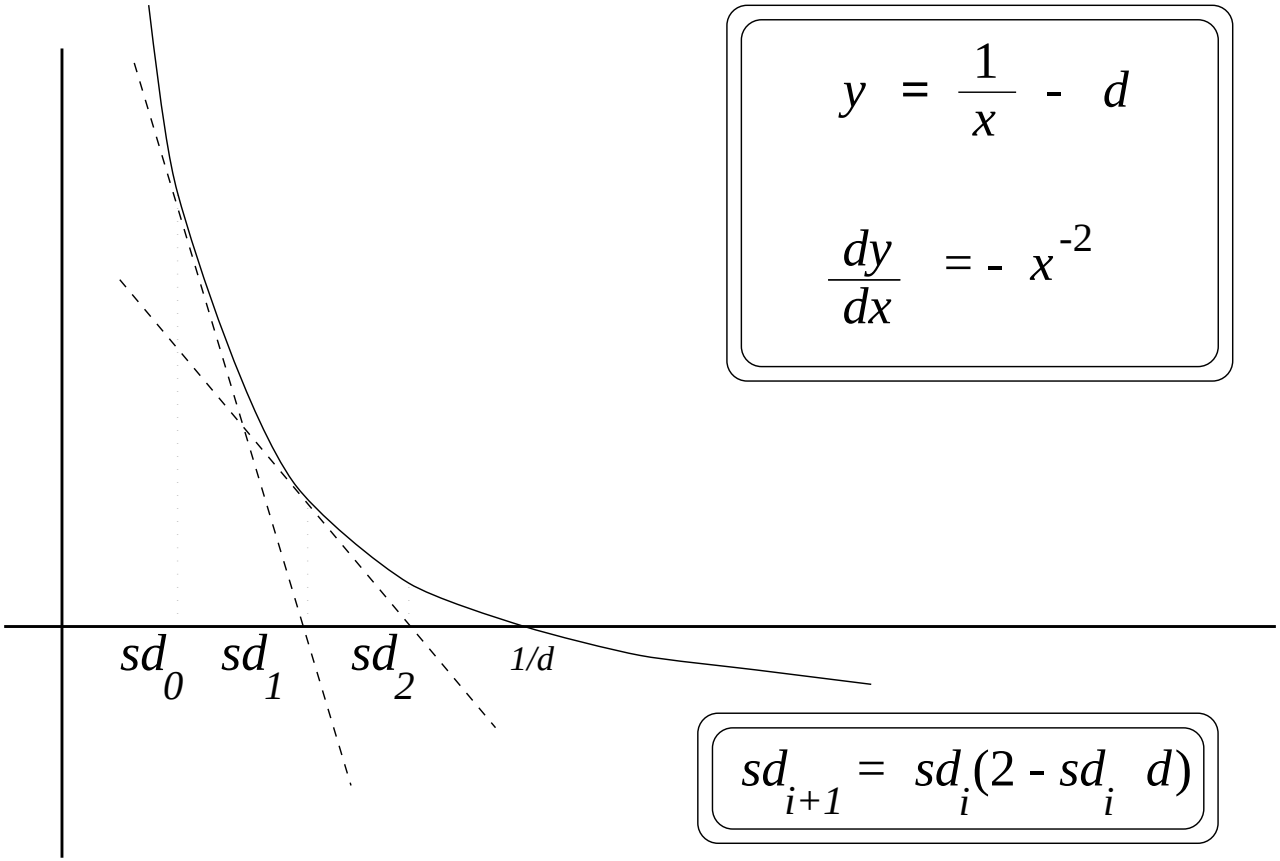
$0.083 \times .0400 = .0033200 \approx .003400 = q_2$

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

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.

To test all possible combinations on the fastest machine in the world today would take over 500 billion billion billion (556×10^{27}) years!

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.

1990s

- FDIV on AMD K5 (Moore-Kaufmann-Lynch)
- AMD Athlon floating point (Russinoff-Flatau)
- Motorola 68020 and Berkeley C String Library (Yu)

1990s

- ...
- Motorola CAP DSP (Brock)
- IBM 4758 secure co-processor (Austel)
- Union Switch and Signal safety-critical checker (Bertolli)

1990s

- ...
- Rockwell Collins microarchitectural equivalence
- Rockwell Collins / aJile Systems JEM1 (Hardin-Greve-Wilding)

2000s

- Rockwell Collins AAMP7 Separation Kernel Microcode
- Rockwell Collins/Green Hills OS Kernel
- Sun Microsystems JVM
- ...

JVM Operational Semantics

M6 is an ACL2 model of the Sun JVM.

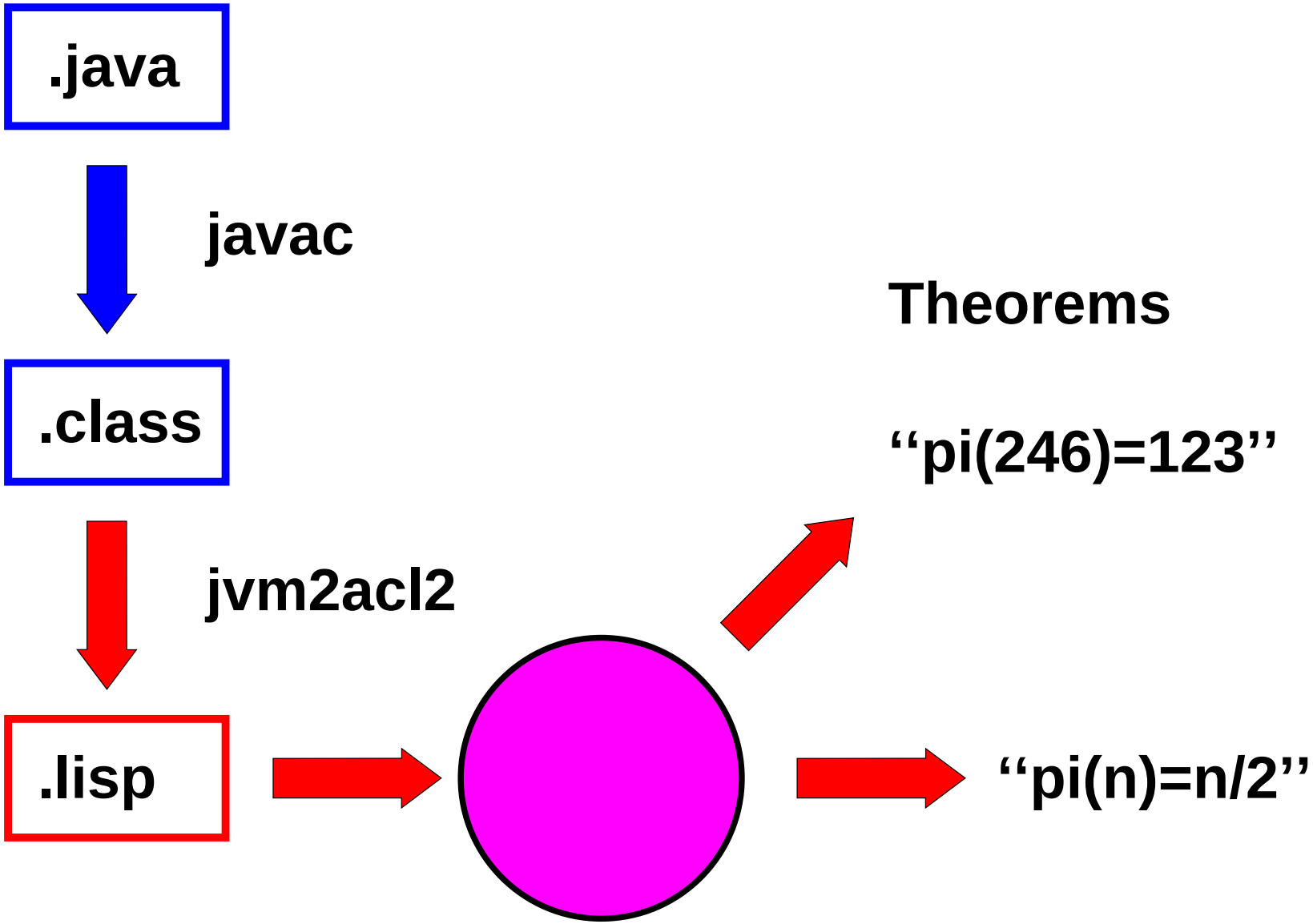
(M6 is a JVM bytecode interpreter written in pure Lisp.)

It executes most J2ME Java programs (except those with significant I/O or floating-point).

M6 was created by Hanbing Liu (now at AMD) with support from Sun Microsystems.

M6 supports

- all data types (except floats),
- multi-threading,
- dynamic class loading,
- class initialization, and
- synchronization via monitors.


.java
javac
.class
jvm2acl2
.lisp
Theorems
“pi(246)=123”
“pi(n)=n/2”

The state we model includes an "external class table" where classes reside until they are loaded.

We have translated the entire Sun CLDC API library implementation into our external representation (672 methods in 87 classes).

The model is 160 pages of ACL2.

ACL2 Demo 2

Present - Why are we succeeding?

Reason 1: Our mathematical logic is an executable programming language.

- Many very efficient heavy-duty implementations
- Supported on many platforms

- Many independently provided programming/system development tools and environments.

Reason 2: We have invested 30 years

- supporting efficient execution,
- integrating a wide variety of proof techniques,
- engineering for industrial scale formulas, and
- developing reusable books.

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

- are bit- and cycle-accurate, not "toys",
- are useful as pre-fab simulation engines, and
- permit mathematical abstraction supported by proof.

Reason 4: Industry has no other alternative; their artifacts are too complicated to analyze accurately any other way.

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.

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