Machines Reasoning About Machines (2015)

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

Boyer-Moore Project

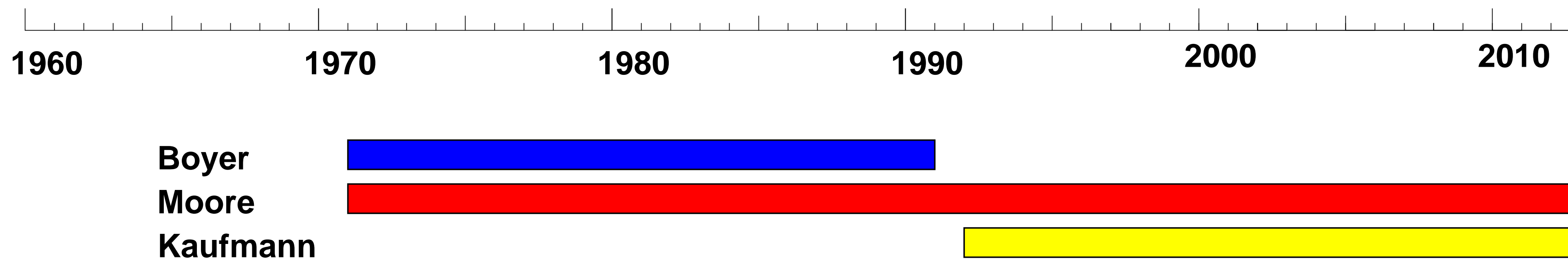
McCarthy's ''Theory of Computation''

Edinburgh Pure Lisp Theorem Prover

A Computational Logic

NQTHM

ACL2



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

Boyer-Moore Project

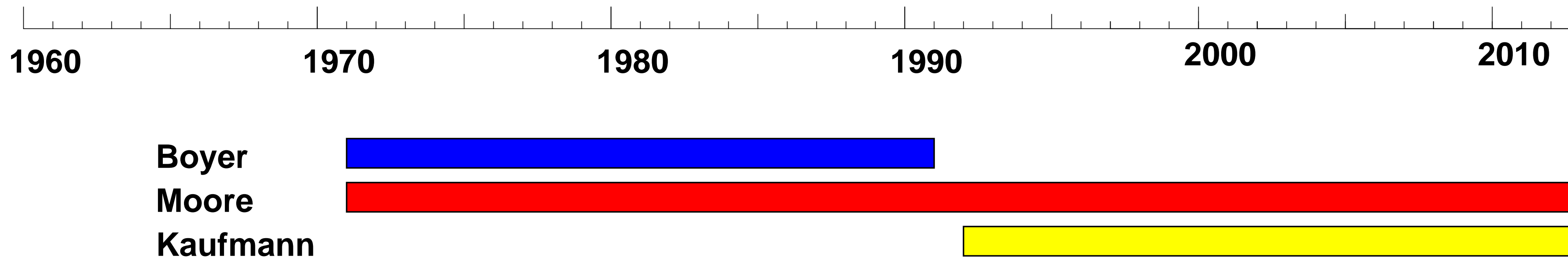
McCarthy’s ‘‘Theory of Computation’’

Edinburgh Pure Lisp Theorem Prover

A Computational Logic

NQTHM

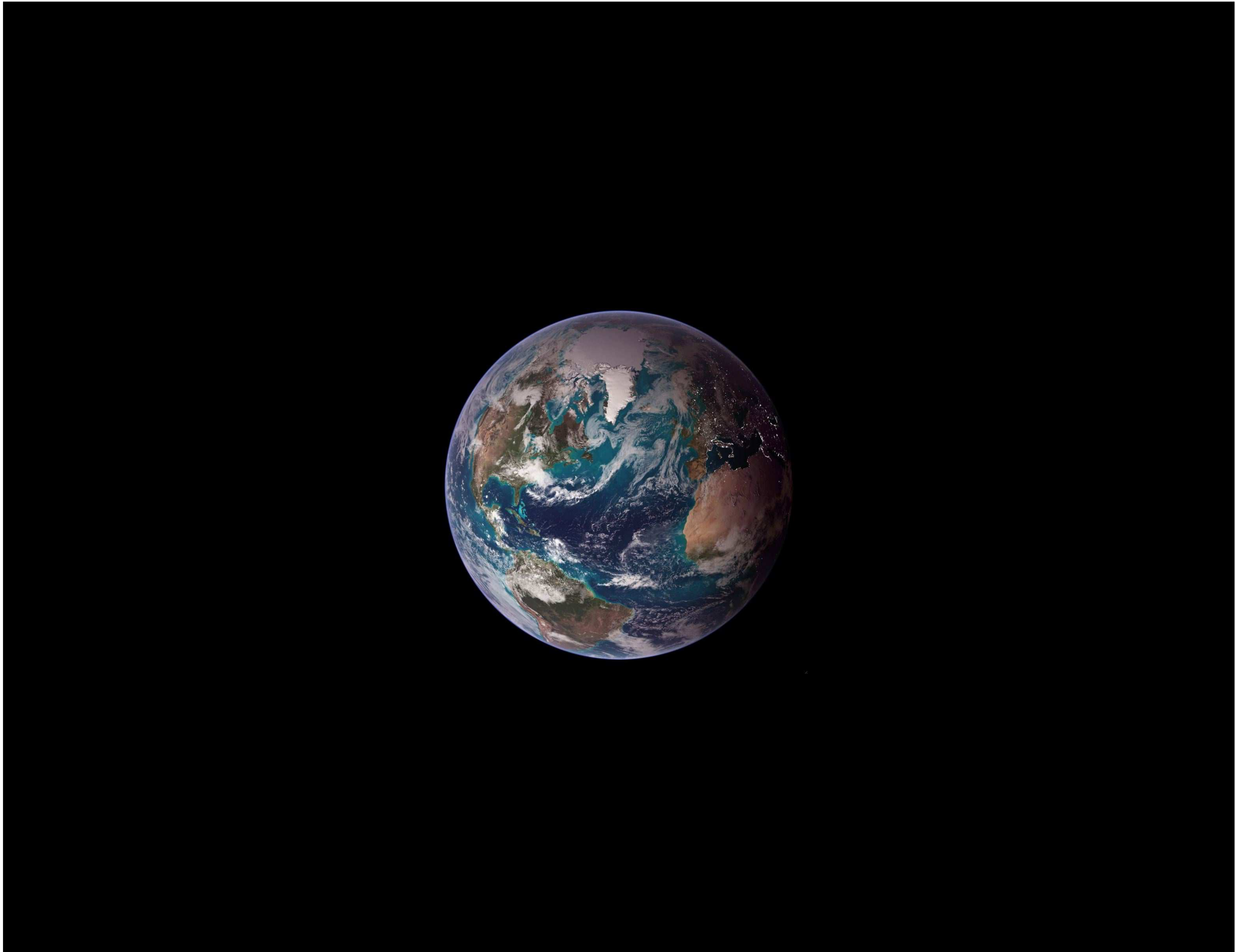
ACL2



How Far Can You Walk?

At 5 km/hour, 8 hours per day, 5 days per week, for the duration of a typical research grant (2 years), you move about 20,000 km.

In 45 years, you move over 400,000 km.



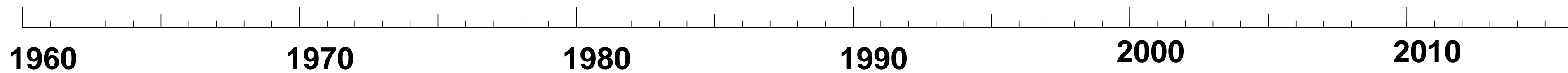
Theorems Proved

simple list processing

academic math and cs

breakthrough commercial applications

regular commercial applications



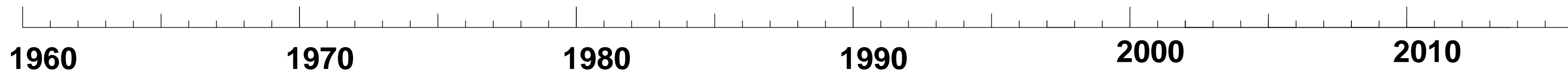
Theorems Proved

simple list processing

academic math and cs

breakthrough
commercial
applications

regular
commercial
applications



A Few Axioms

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

ACL2 includes primitives for integers, rationals, complex rationals, conses, symbols, characters, and strings.

Theorems Proved: 1970s

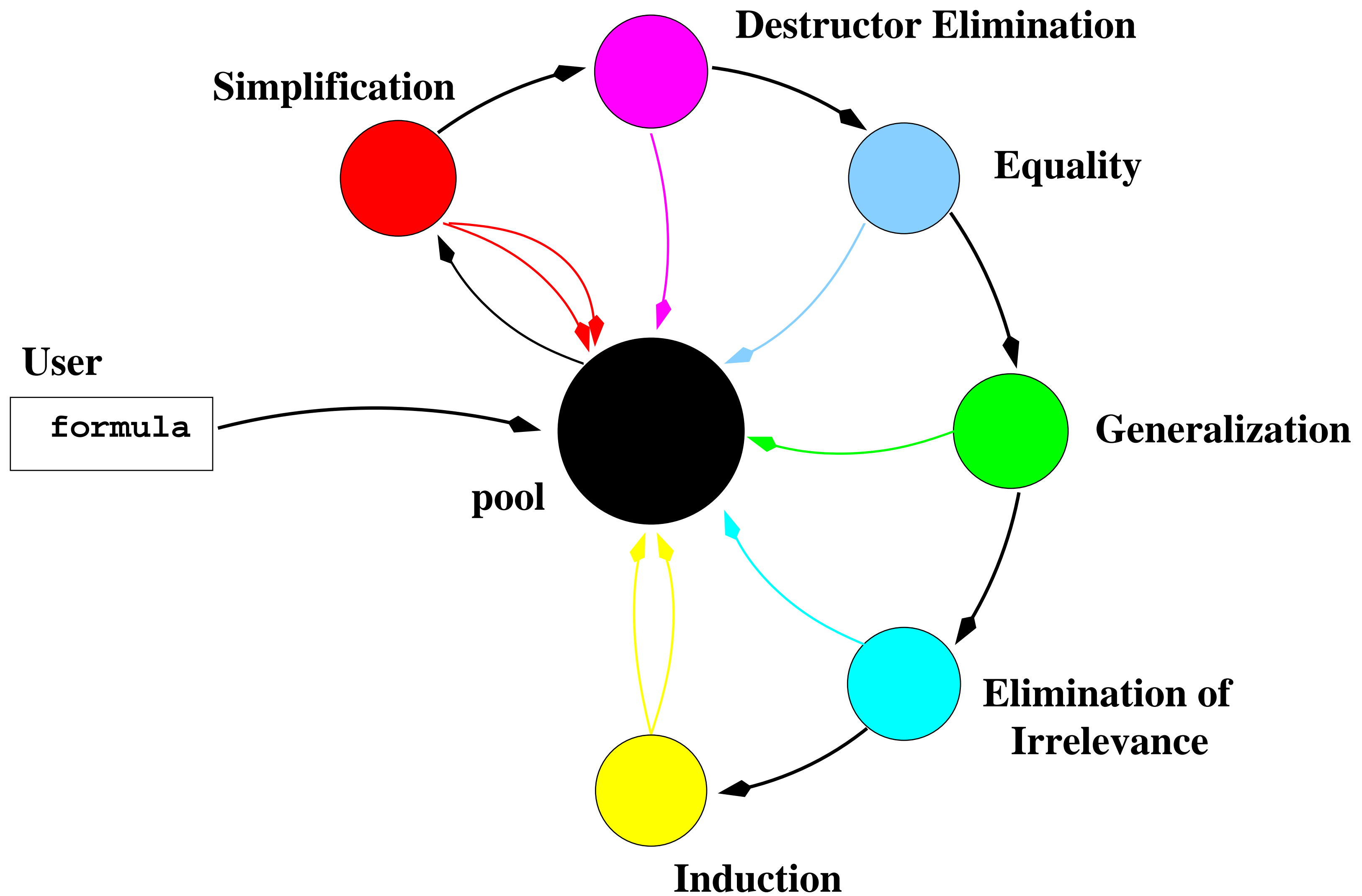
- ap is associative:

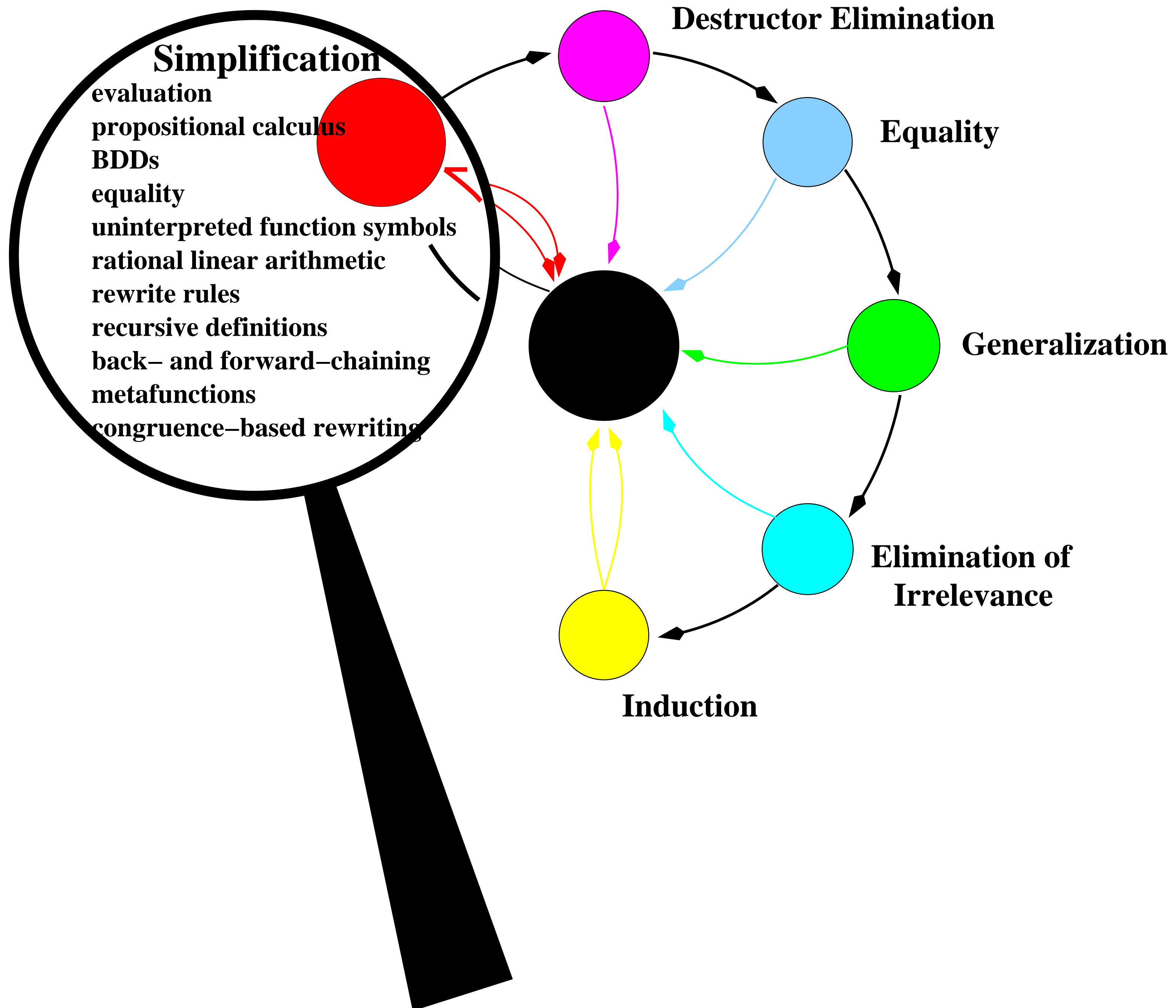
```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

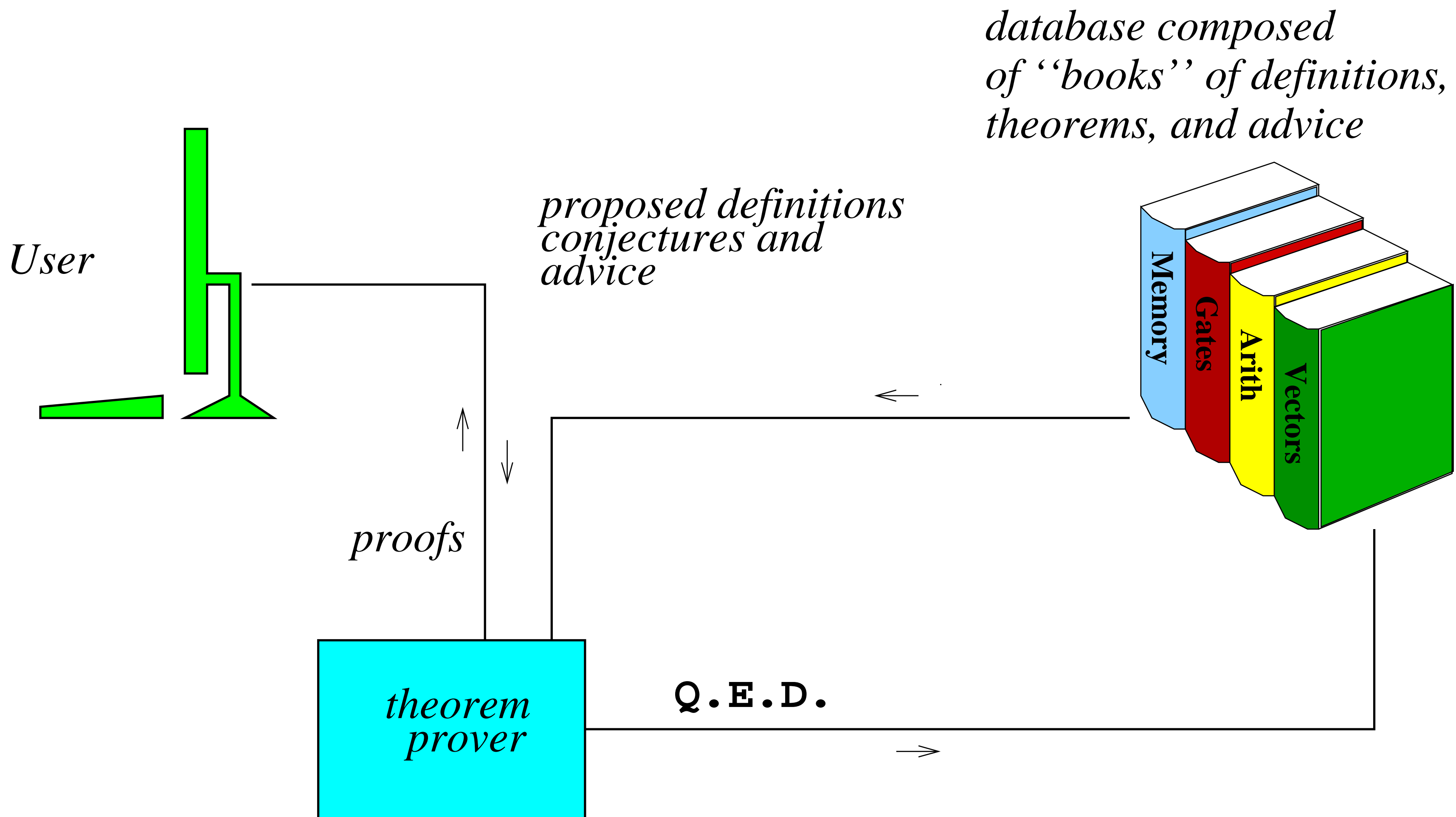
$\forall a \forall b \forall c :$ $\mathsf{ap}(\mathsf{ap}(a,b),c) = \mathsf{ap}(a,\mathsf{ap}(b,c))$.

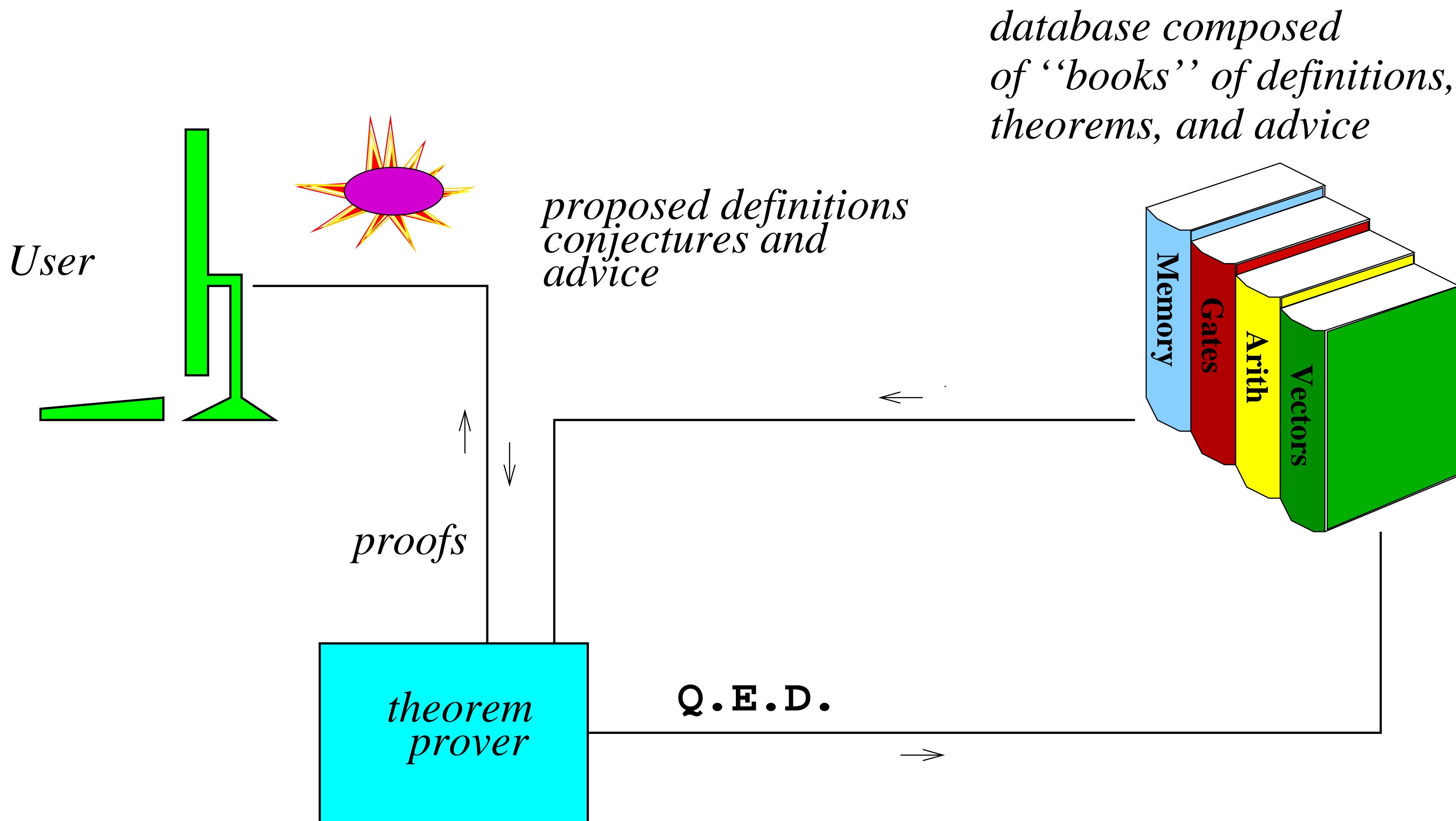
Definition

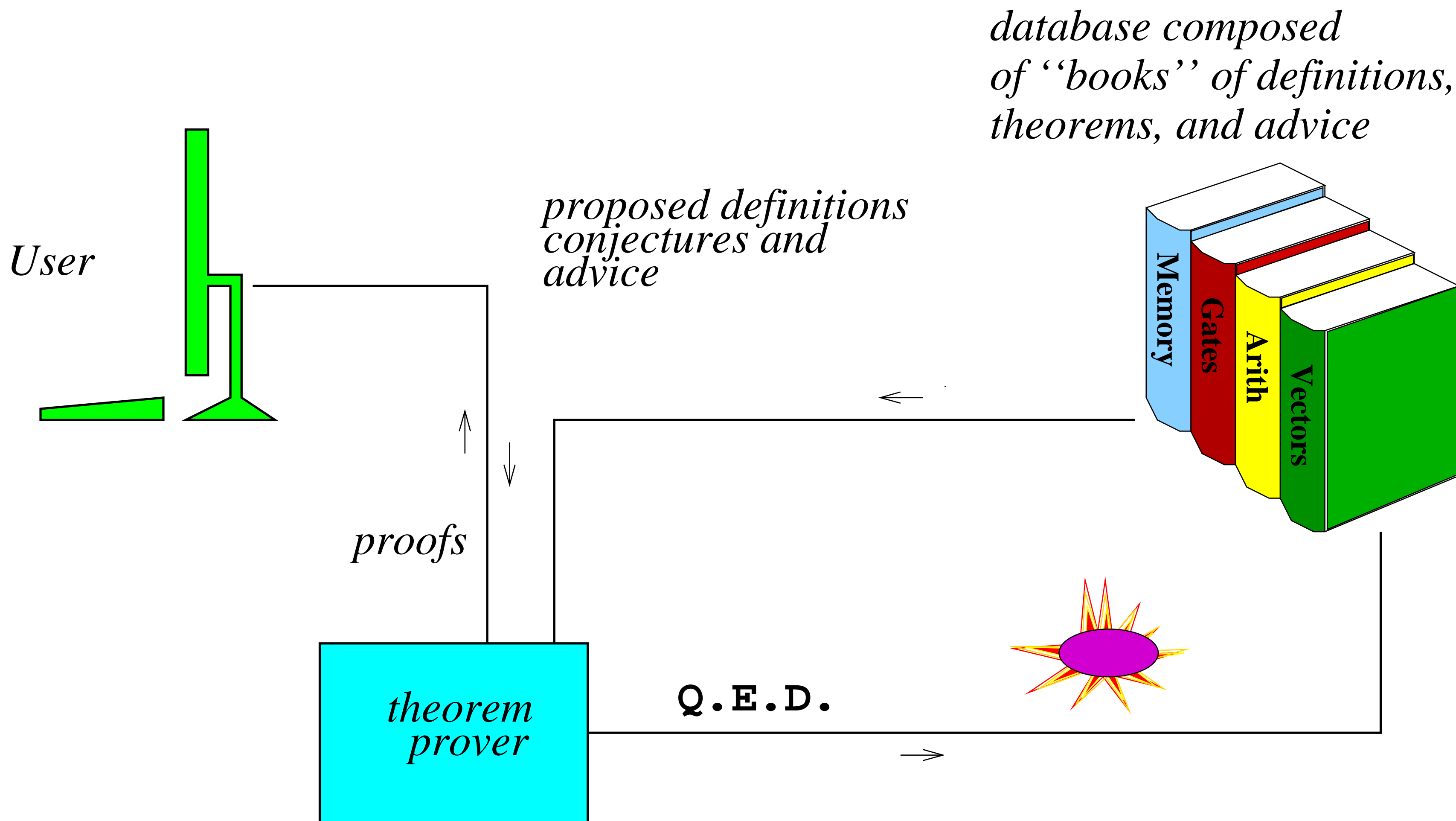
```
(defun ap (x y)
  (if (endp x)
      y
      (cons (car x)
            (ap (cdr x) y))))

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


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

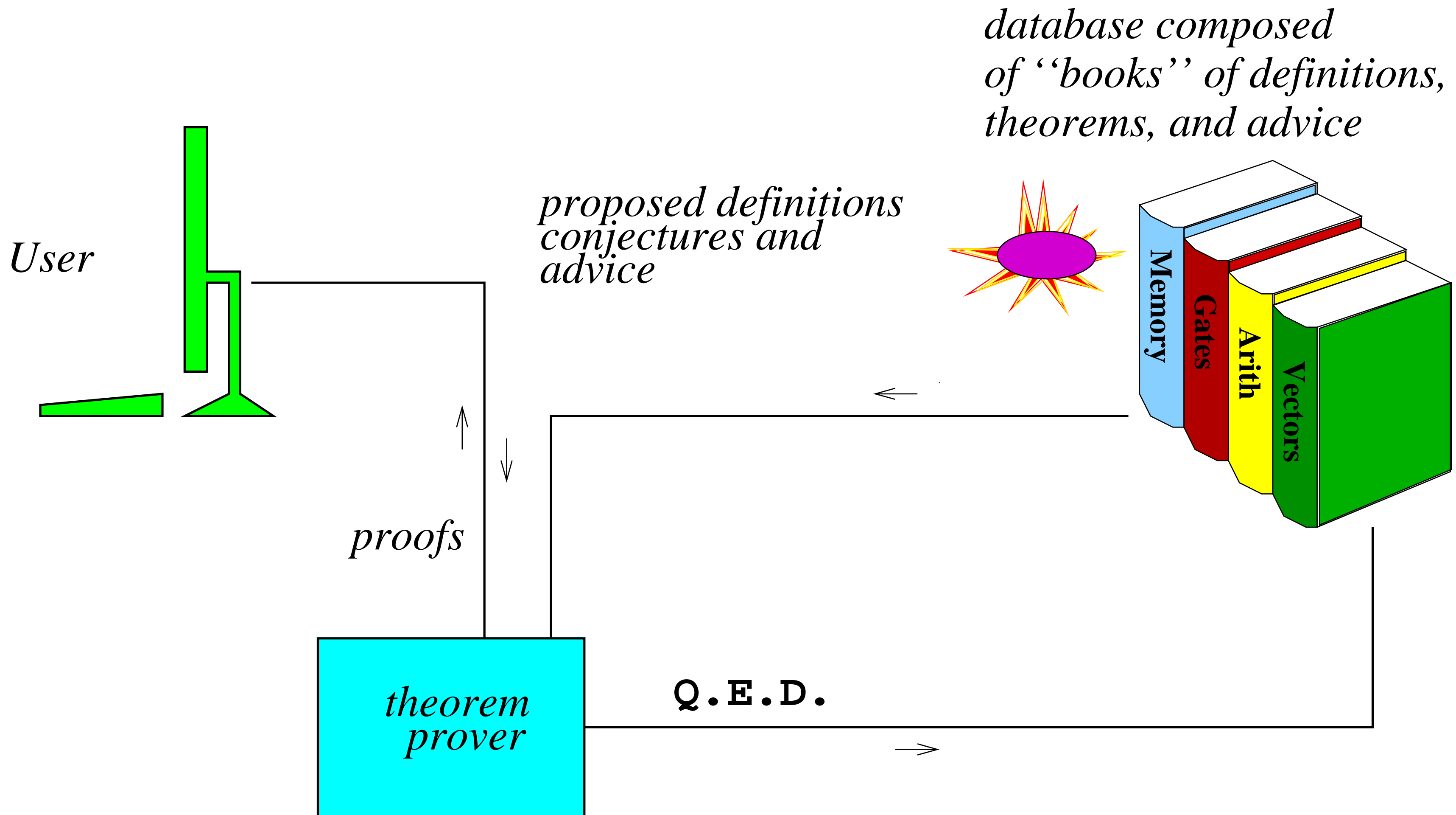

Destructor Elimination
Equality
Generalization
Elimination of
Irrelevance
Induction
Simplification
evaluation
propositional calculus
BDDs
equality
uninterpreted function symbols
rational linear arithmetic
rewrite rules
recursive definitions
back– and forward–chaining
metafunctions
congruence–based rewriting


database composed
of ‘‘books’’ of definitions,
theorems, and advice
proposed definitions
conjectures and
advice
User
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
proposed definitions
conjectures and
advice
User
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.

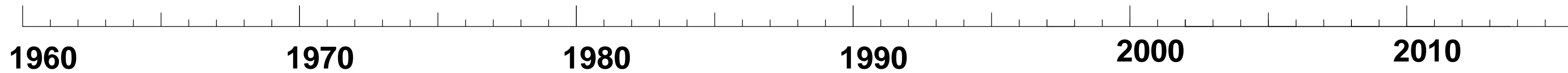
Theorems Proved: 1980s

simple list processing

academic math and cs

breakthrough
commercial
applications

regular
commercial
applications



1980s Academic Math

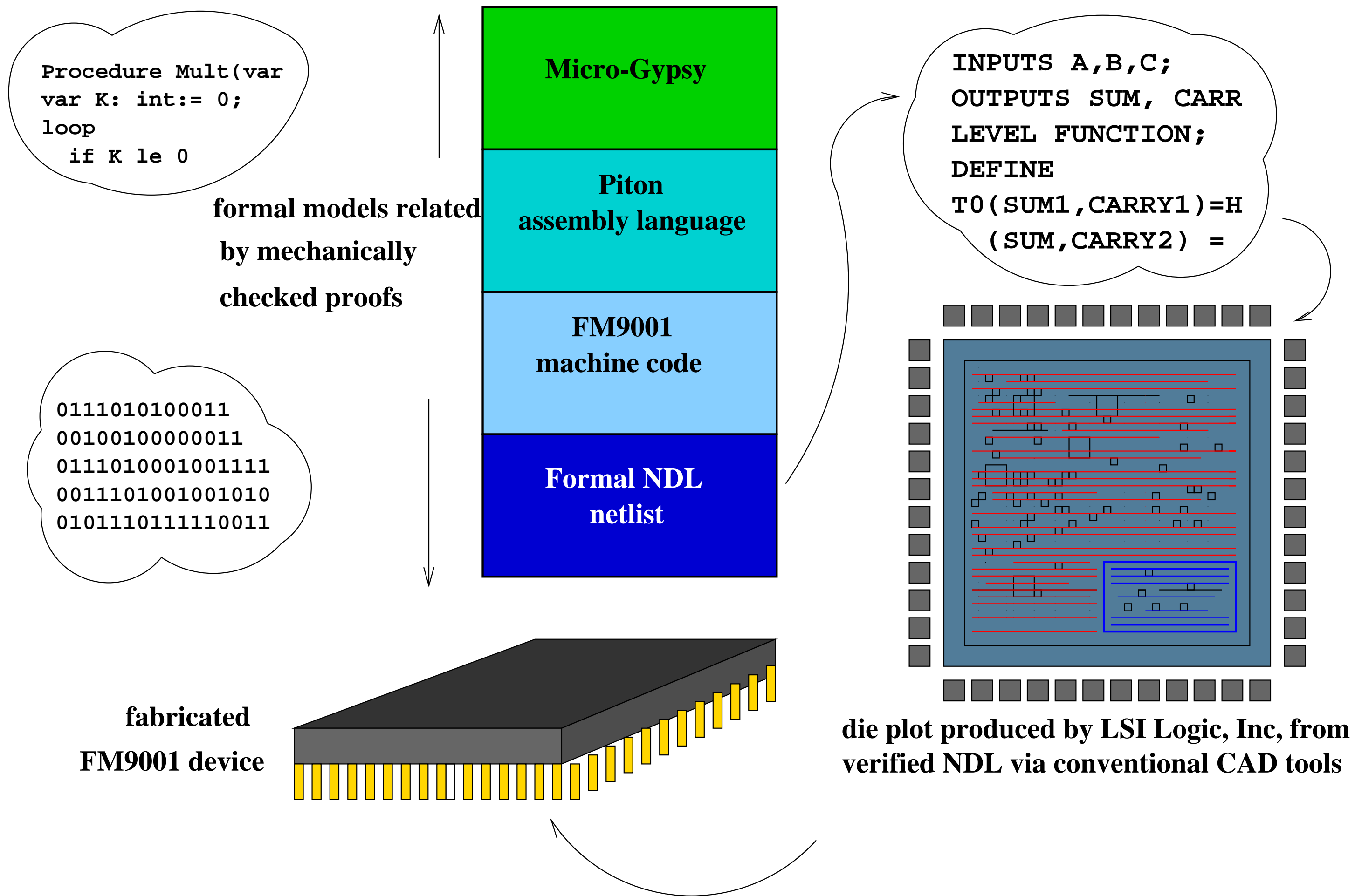
- undecidability of the halting problem (18 lemmas)
- invertibility of RSA encryption (172 lemmas)

- Gauss' law of quadratic reciprocity
  [Russinoff]
  (348 lemmas)

- Gödel's First Incompleteness Theorem
  [Shankar]
  (1741 lemmas)

1980s Academic CS

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



Procedure Mult(var
var K: int:= 0;
loop
if K le 0
Micro-Gypsy
Piton
assembly language
formal models related
by mechanically
checked proofs
FM9001
machine code
0111010100011
00100100000011
0111010001001111
0011101001001010
0101110111110011
Formal NDL
netlist
INPUTS A,B,C;
OUTPUTS SUM, CARR
LEVEL FUNCTION;
DEFINE
T0(SUM1,CARRY1)=H
(SUM,CARRY2) =
fabricated
FM9001 device
die plot produced by LSI Logic, Inc, from
verified NDL via conventional CAD tools

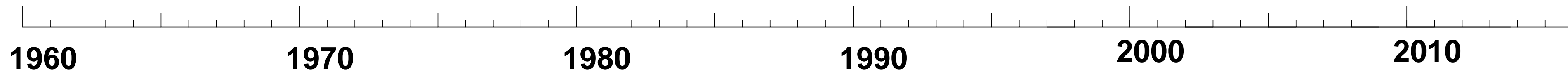
Theorems Proved: 1990s

simple list processing

academic math and cs

breakthrough
commercial
applications

regular
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]`
...	...	$= \ldots$	...		
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
      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:

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

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

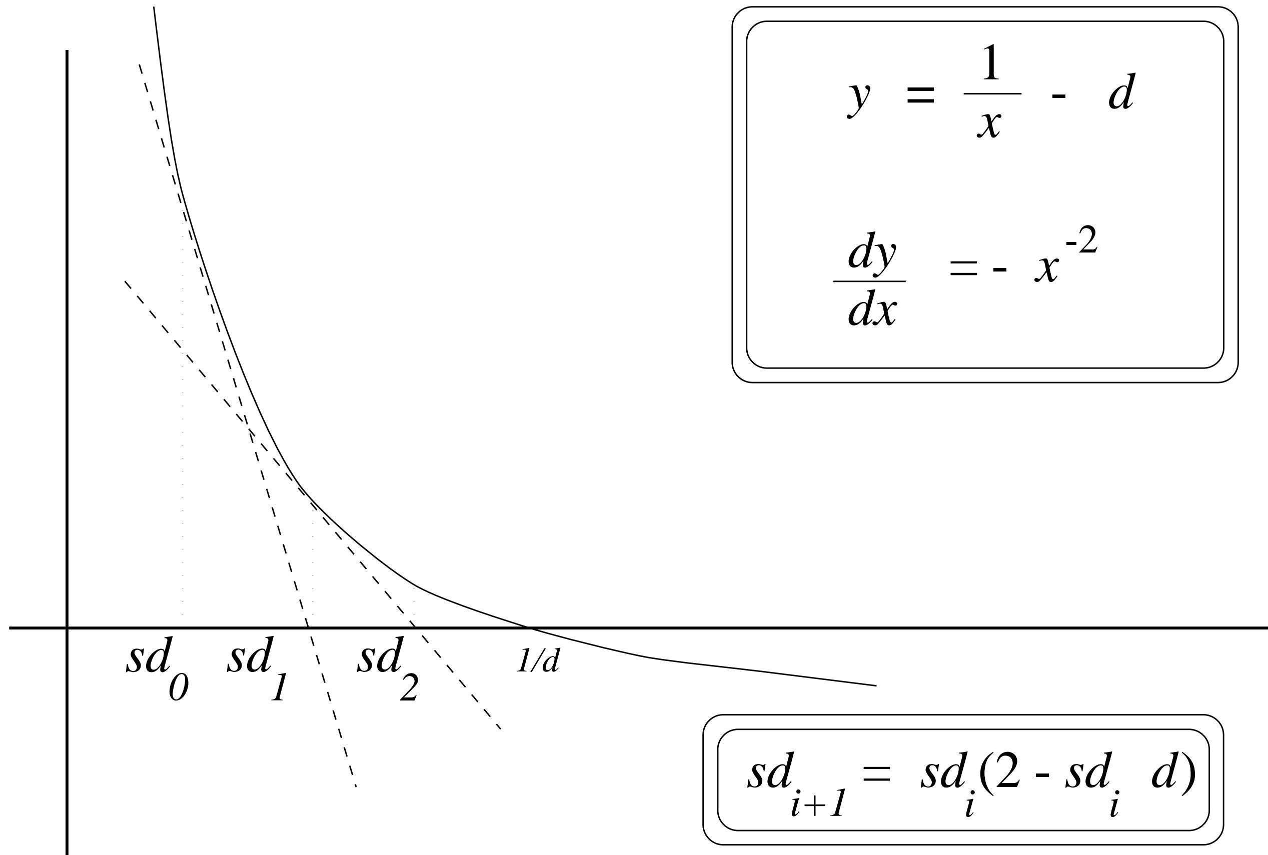
$0.083 \times .0400 = .0033200 \approx .003400 = q_2$

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

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

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 Tianhe-2 (33.86 petaflops), testing all possible cases would take

8178337571240167641483597

$\sim 8 \times 10^{24}$ *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*)

Changing the Process

AMD implemented a translator from RTL to ACL2, and

compared it with their RTL emulator on hundreds of millions of floating point tests. It was “bit- and cycle-accurate.”

All elementary floating point operations on the AMD Athlon and Opteron have been mechanically verified with ACL2.

Aside: Hardware v Software Verification

“Is ACL2 only good for hardware verification?”

All ACL2 models and specifications are written in (a subset of) an ANSI standard programming language: Common Lisp.

Thus, everything we do in ACL2 can be understood as software verification.

Hardware companies are more amenable to formal verification because specifications are clearer, designers are committed to specs, and post-silicon bugs are expensive.

1990s

- FDIV on AMD K5 (Moore-Kaufmann-Lynch)
- AMD Athlon floating point (Russinoff-Flatau)
- Motorola 68020 binaries produced by gcc −o for Berkeley C String Library (Yu)

1990s

- ...
- Motorola CAP DSP (Brock)
- Rockwell Collins microarchitectural equivalence
- Rockwell Collins / aJile Systems JEM1 (Hardin-Greve-Wilding)

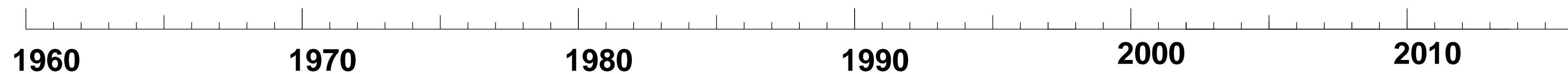
Theorems Proved: 1990s

simple list processing

academic math and cs

breakthrough
commercial
applications

regular
commercial
applications



2000s

- IBM Power4 divide and square root (Sawada)
- Rockwell Collins AAMP7 Separation Kernel Microcode
- Rockwell Collins/Green Hills OS Kernel
- Sun Microsystems JVM

JVM Operational Semantics

M6 is a JVM bytecode interpreter in ACL2.

Built to verify properties of JVM code for class loading.

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.

The model is 160 pages of ACL2 (plus 500 pages of constants representing the CLDC API with 87 classes containing 672 methods).

```
Mac% cat Demo.java
class Demo {
  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;
    }
}

Mac% javac Demo.java
Mac% javap -c Demo
Compiled from "Demo.java"
```

```
class Demo {
  Demo();
    Code:
       0: aload_0
       1: invokespecial #1   // Method java/lang/Object."<init>":()V
       4: return

  public static int fact(int);
    Code:
       0: iload_0
       1: ifle          13
       4: iload_0
       5: iload_0
       6: iconst_1
       7: isub
       8: invokestatic  #2  // Method fact:(I)I
      11: imul
```

```
      12: ireturn
      13: iconst_1
      14: ireturn

  public static void main(java.lang.String[]);
    Code:
       0: aload_0
       1: iconst_0
       2: aaload
       3: bipush        10
       5: invokestatic  #3  // Method java/lang/Integer.parseInt:(Ljava/lang/Str
       8: istore_1
       9: getstatic     #4  // Field java/lang/System.out:Ljava/io/PrintStream;
      12: iload_1
      13: invokestatic  #2  // Method fact:(I)I
      16: invokevirtual #5  // Method java/io/PrintStream.println:(I)V
      19: return
}
```

```
Mac% java Demo 6
720
```

ACL2 Demo

```
Mac% java Demo 6
720
Mac% java Demo 20
```

```
Mac% java Demo 6
720
Mac% java Demo 20
-2102132736
Mac%
```

Quick Summary of Some Current Projects

- Oracle: FP and JVM
- Intel: FP, SystemC modeling, elliptic curve crypto
- Kestrel: JVM+Android modeling, Android app verification
- AMD: high-level transaction protocol analysis tools

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.

x86 ISA in ACL2

Intended Applications of the x86 model:

- x86 ISA emulation
- System code verification
- Development of trustworthy programs
- Build-to (for x86 vendors) and compile-to (for user) specs

- Goal: booting FreeBSD, verification of user and system programs

Work by Hunt, Goel, *et at*.

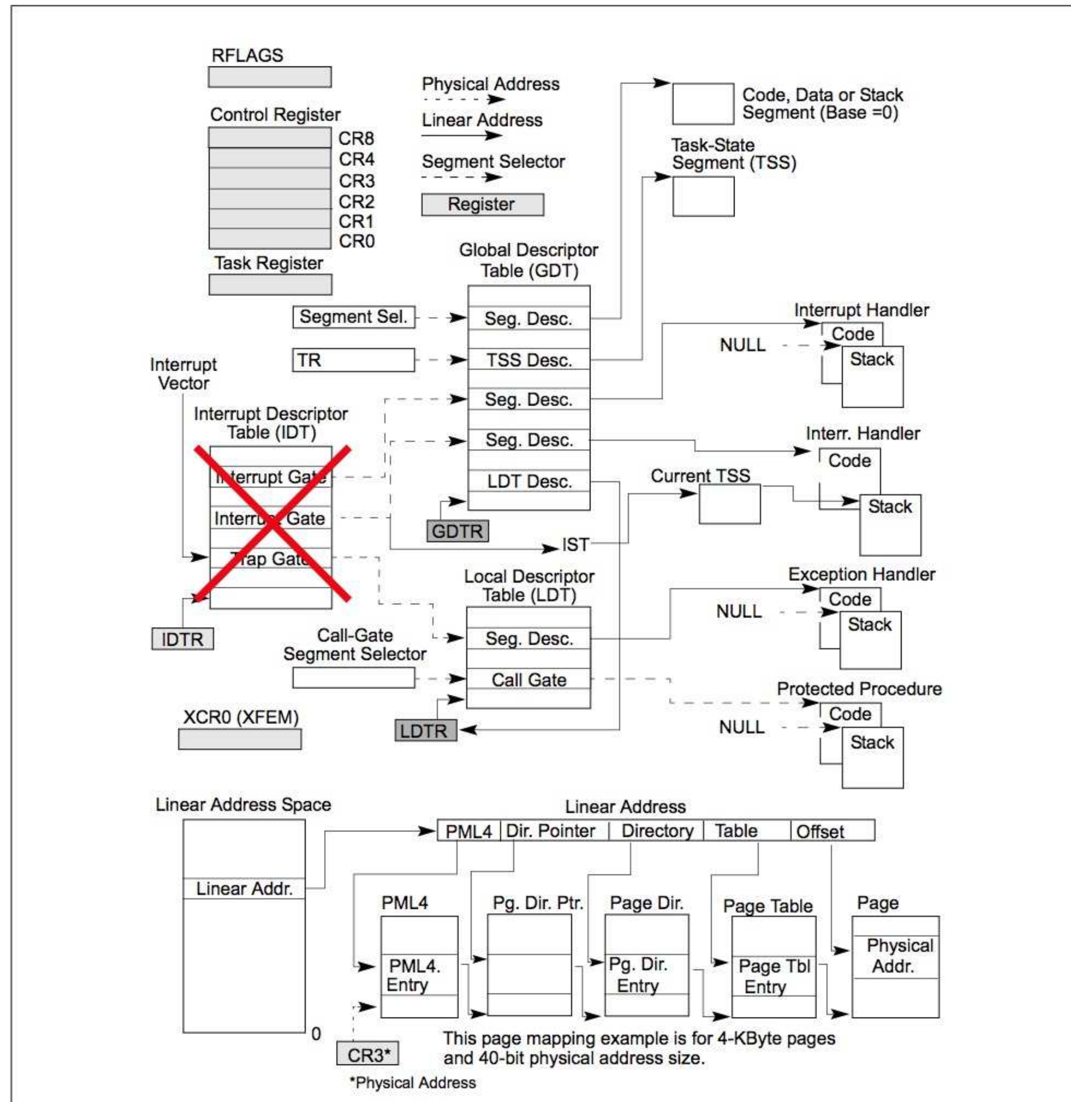


Figure 2-2. System-Level Registers and Data Structures in IA-32e Mode

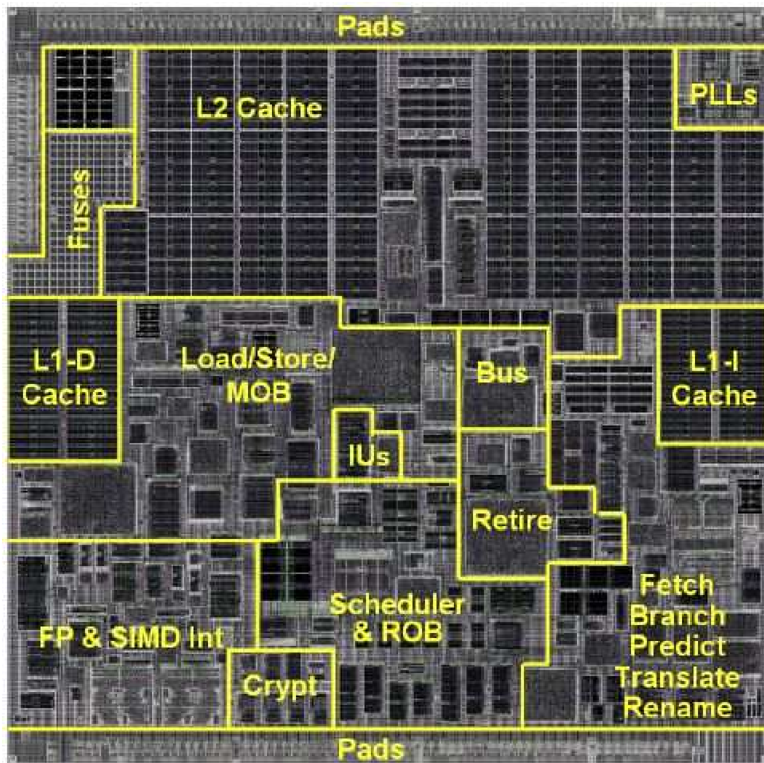
x86 ISA in ACL2

Performance

user level: $\sim$ 3.3 million ips

system level: $\sim$ 912,000 ips

Centaur (VIA) Nano 64-bit x86 CPU



- ∼20 FP operations verified
- LOC: FADD is 33,700 lines of Verilog
- Proof Times: few secs – few hrs
- Memory: 1 – 80 GB
- design and (automatic) proofs change daily
- hard-to-discover bugs found (e.g., 4 tests out of 2^{160} would have failed)

Centaur (VIA) QuadCore

4 Nano 64-bit x86

“most efficient x86 multicore processor on the planet”

Lenova and HP use QuadCore chips in all laptops.

ACL2 is now a critical part of the workflow (after 8 years of investment in ACL2 use at Centaur).

E.g., at Centaur:

Module Browser

Linter

VL parsetree

SV model

ACL2

GL

Equivalence Checker

Proof!

SystemVerilog RTL

Bit Blast

AIG formula

```
(defun operand-spec
   (op1 op2)
   ...)
```

SAT solver

CenTaur Technology

Highly-Automated Process

- automatic translation (by an ACL2 program) of 1.1M lines of Verilog into a form suitable for ACL2 analysis
  - nightly regression of all proofs (run on 100s of machines)
  - errors introduced yesterday are addressed today
  - automated low-level reasoning using ACL2 verified bit-blasting

– ACL2 verified tools check wire-level engineering changes and clock trees (saving days)
– cost of ACL2 staff is less than license fees previously paid for regression testing

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 45 years

- supporting efficient execution *and* proof (so models are dual-purpose)
- integrating a wide variety of proof techniques (so proofs are more automatic)
- engineering for industrial scale formulas
- developing reusable books

- interfacing to other tools (e.g., IBM Sixth Sense, ABC, SAT, MC) (so embedded theorem proving can glue disparate fragments together), and
- supporting verification tool building (so users can build, verify, and then efficiently execute special-purpose tools)

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: We have a *very* talented user community, without whom a talk like this would be impossible.

Reason 4: We have a *very* talented user community, without whom a talk like this would be impossible.

> "The reason the Boyer-Moore theorem is so 'good' is that only smart people use it!" – *anonymous critic, early 1980s*

Reason 5: Industry has no other alternative than to use mechanized reasoning; 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

- *decreases* time-to-market, and
- *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

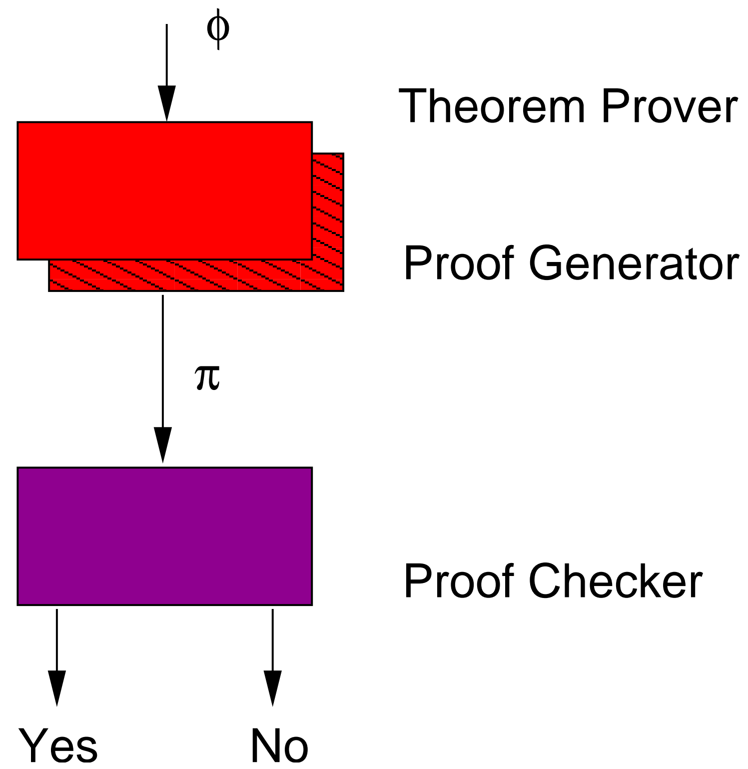
How Do We Know ACL2 is Sound?

“Trust us!” – *Kaufmann and Moore*

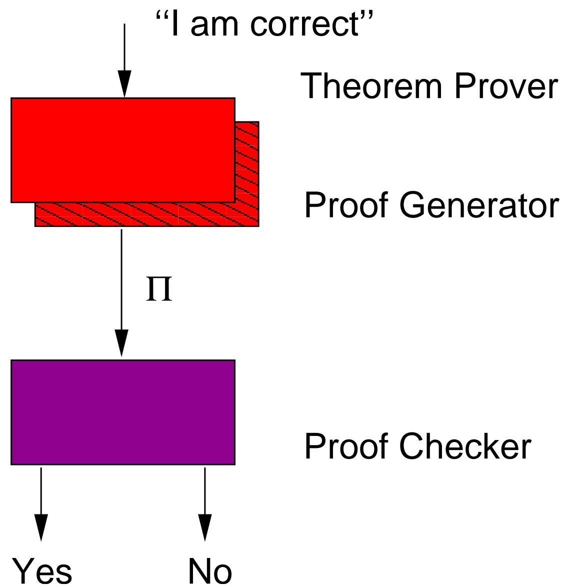
Obviously, we would like to prove it correct.

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"