

# **A Theorem Proving Tool for Program Analysis: ACL2**

J Strother Moore  
(with Matt Kaufmann)  
Department of Computer Sciences  
University of Texas at Austin

# Quick Summary

- **System Name and Contact Details:**

**ACL2** (**A C**omputational **L**ogic for  
**A**pplicative **C**ommon **L**isp)

{kaufmann,moore}@cs.utexas.edu

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

- **Logic:** Quantifier-Free First Order Logic with Induction (think of Pure Lisp without `apply`)
- **Decidability:** Undecidable – but ACL2 is complete on certain fragments:  
propositional calculus, equality and uninterpreted function symbols, rational linear arithmetic

- **Theorem Proving Paradigm:** Rewriting using previously proved lemmas, decision procedures, and automatic induction
- **User Interaction Paradigm:** User “programs” the system’s behavior by proving lemmas

- **Typical Applications:**

- functional correctness of commercial floating point units (AMD, Centaur, IBM)
- functional correctness of microprocessors and virtual machines (Rockwell Collins, Centaur, Sun)

- information flow and isolation of OS kernels (Rockwell Collins)
- other properties of such commercial artifacts
- algorithm correctness
- meta-mathematical results (e.g., correctness of computer algebra algorithms, proof checkers, and theorem provers)

# Software Verification

In ACL2, all semantic models and specifications are Common Lisp programs

Thus, everything we do in ACL2 is *software verification*

ACL2 is an integrated verification environment for a practical subset of an ANSI standard programming language

## **ACL2 is Written in ACL2**

ACL2 is written in applicative Common Lisp (ACL2). It is probably one of the largest functional software systems in use

## System Statistics (V4.2)

<i>type</i>	<i>KLOC</i>	<i>MB</i>	
source code	94	4.1	
comments	51	3.0	
documentation	78	3.5	(4.8MB HTML)
total	244	10.6	

## Regression Suite Statistics (V4.2)

number of functions defined	31,940
number of theorems proved	83,962
number of files	2,825
KLOC	1,410
MB	45

# Boyer-Moore Project

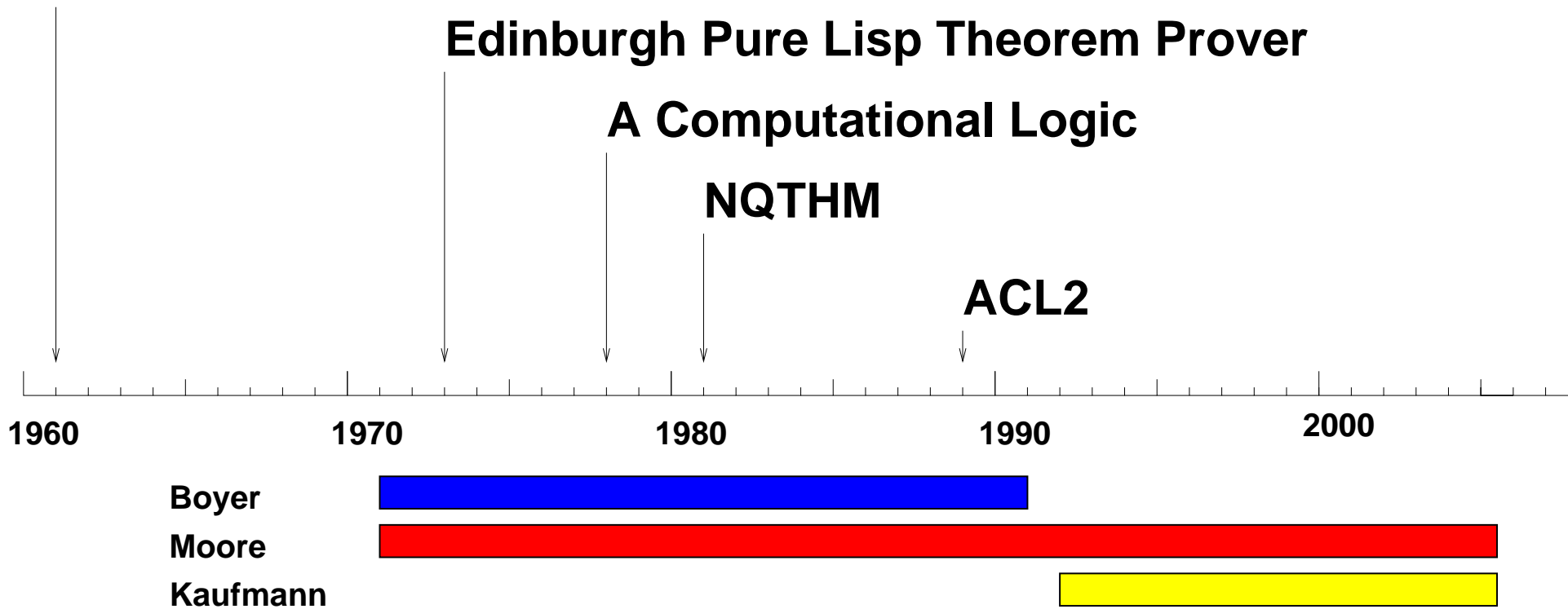
McCarthy's "Theory of Computation"

Edinburgh Pure Lisp Theorem Prover

A Computational Logic

NQTHM

ACL2

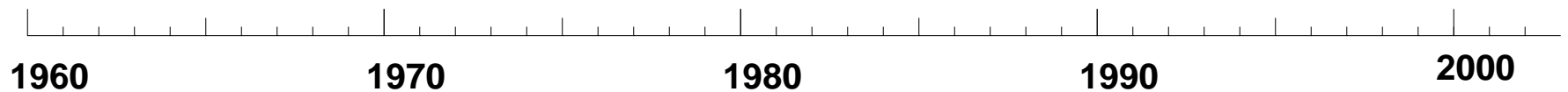


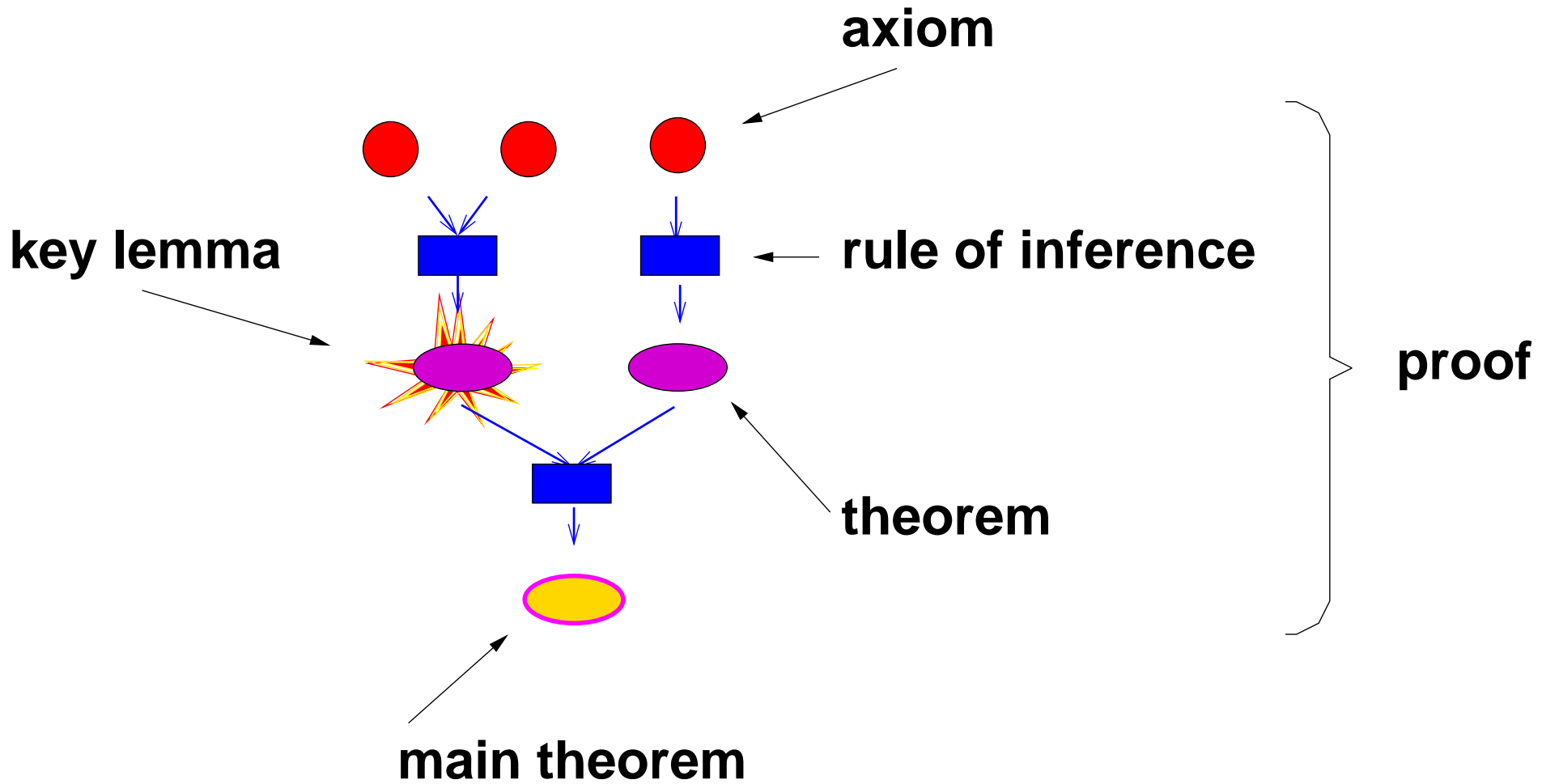
# Theorems Proved

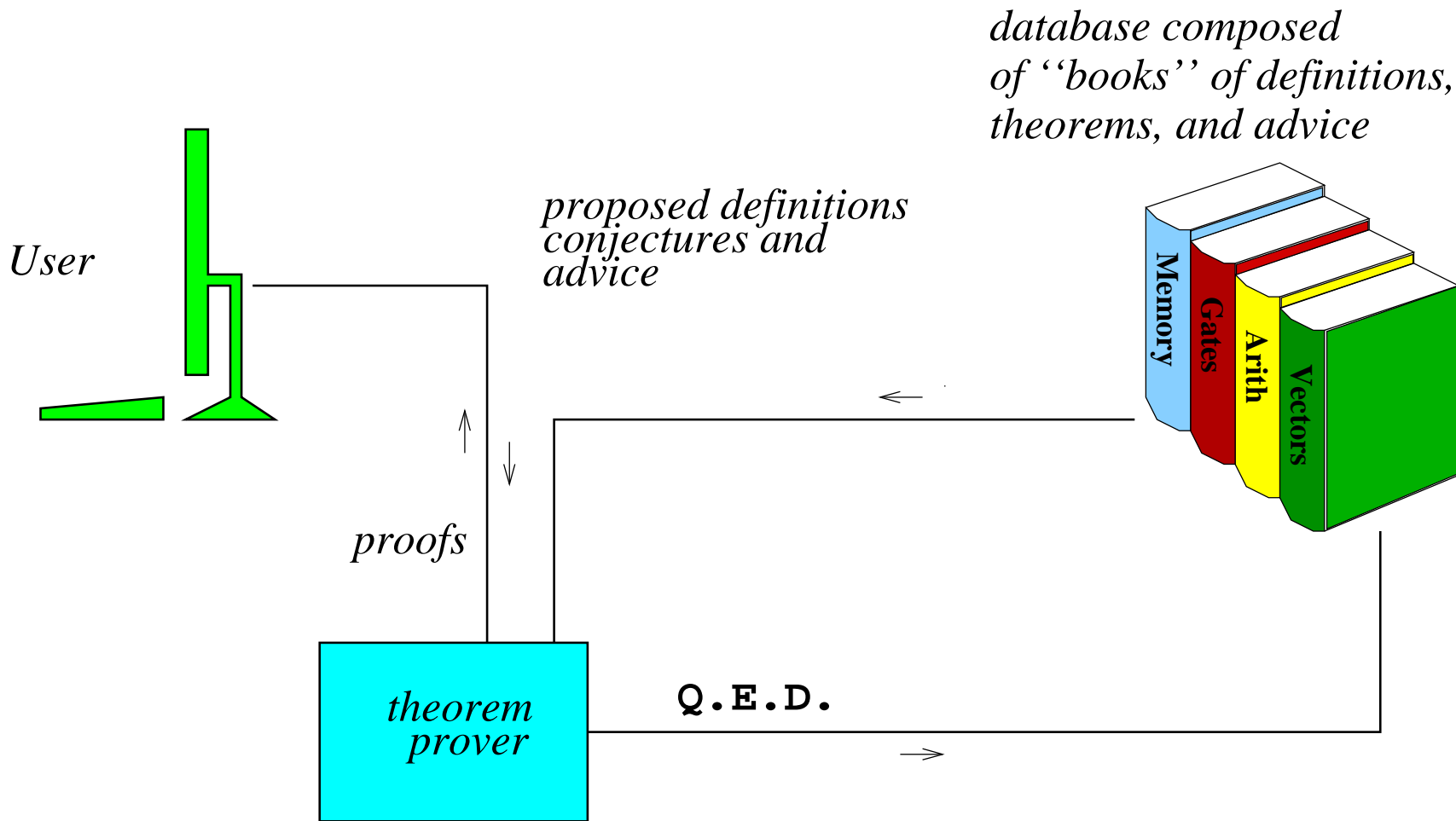
**simple list processing**

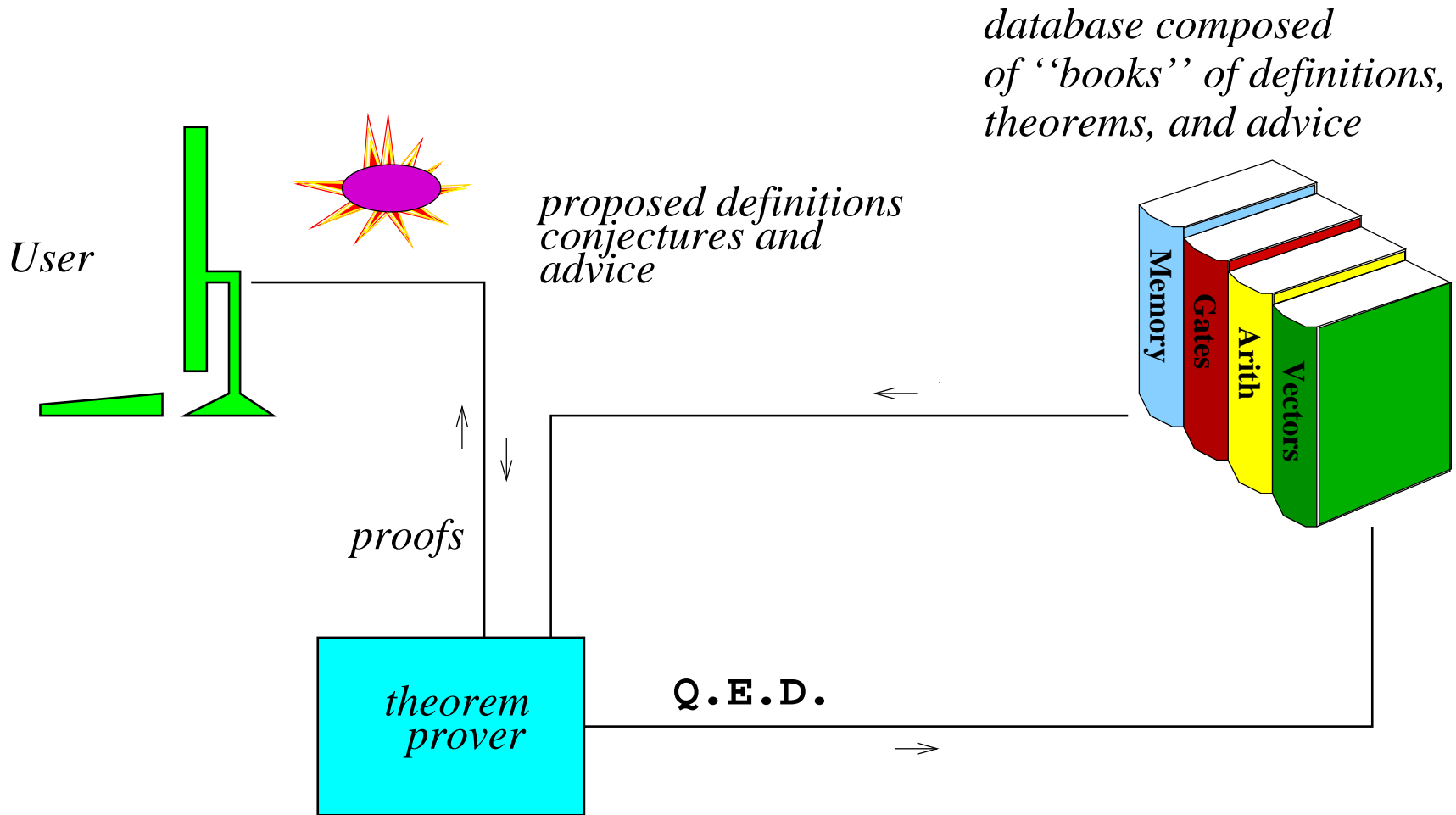
**academic math and cs**

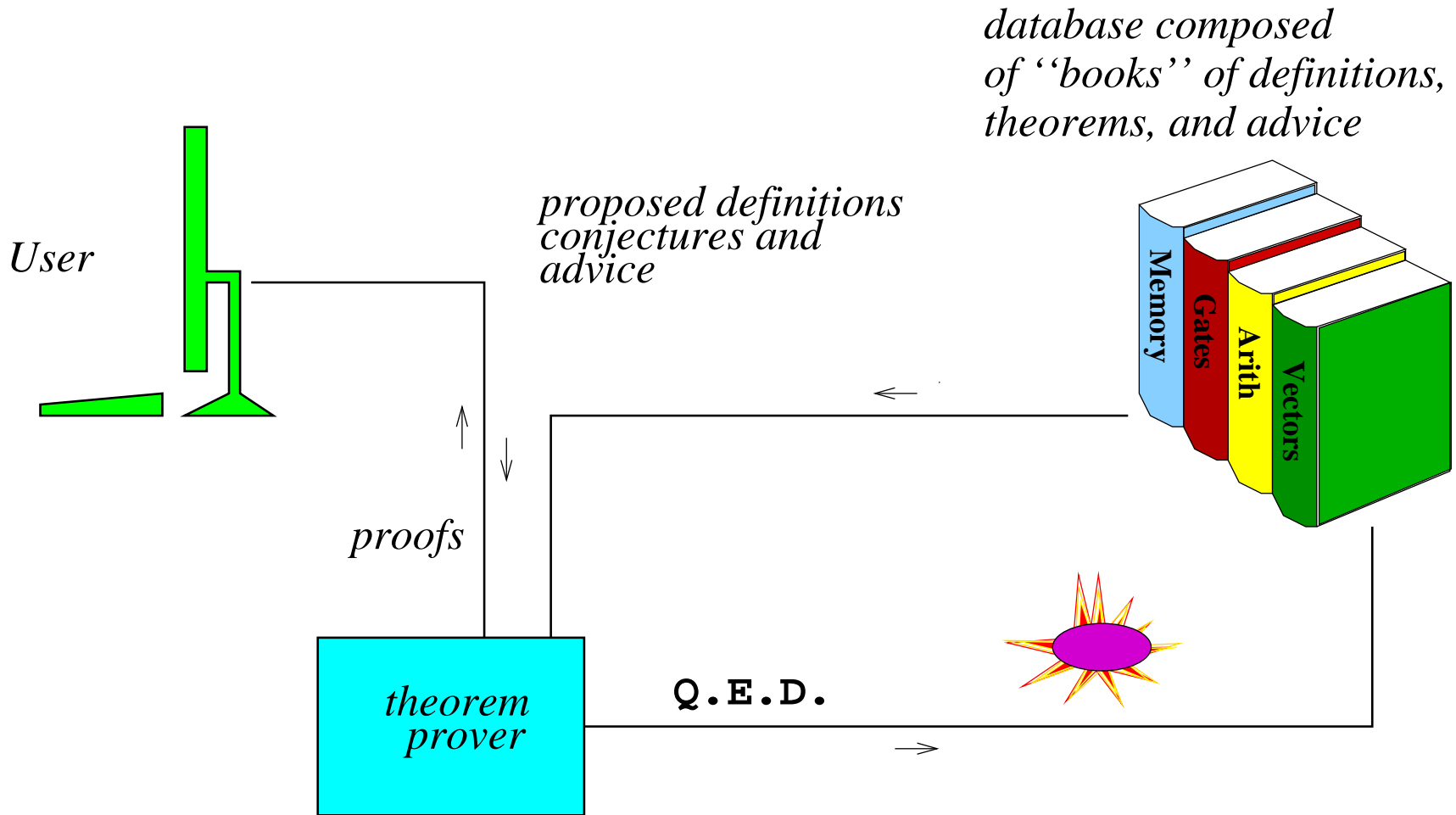
**commercial  
applications**

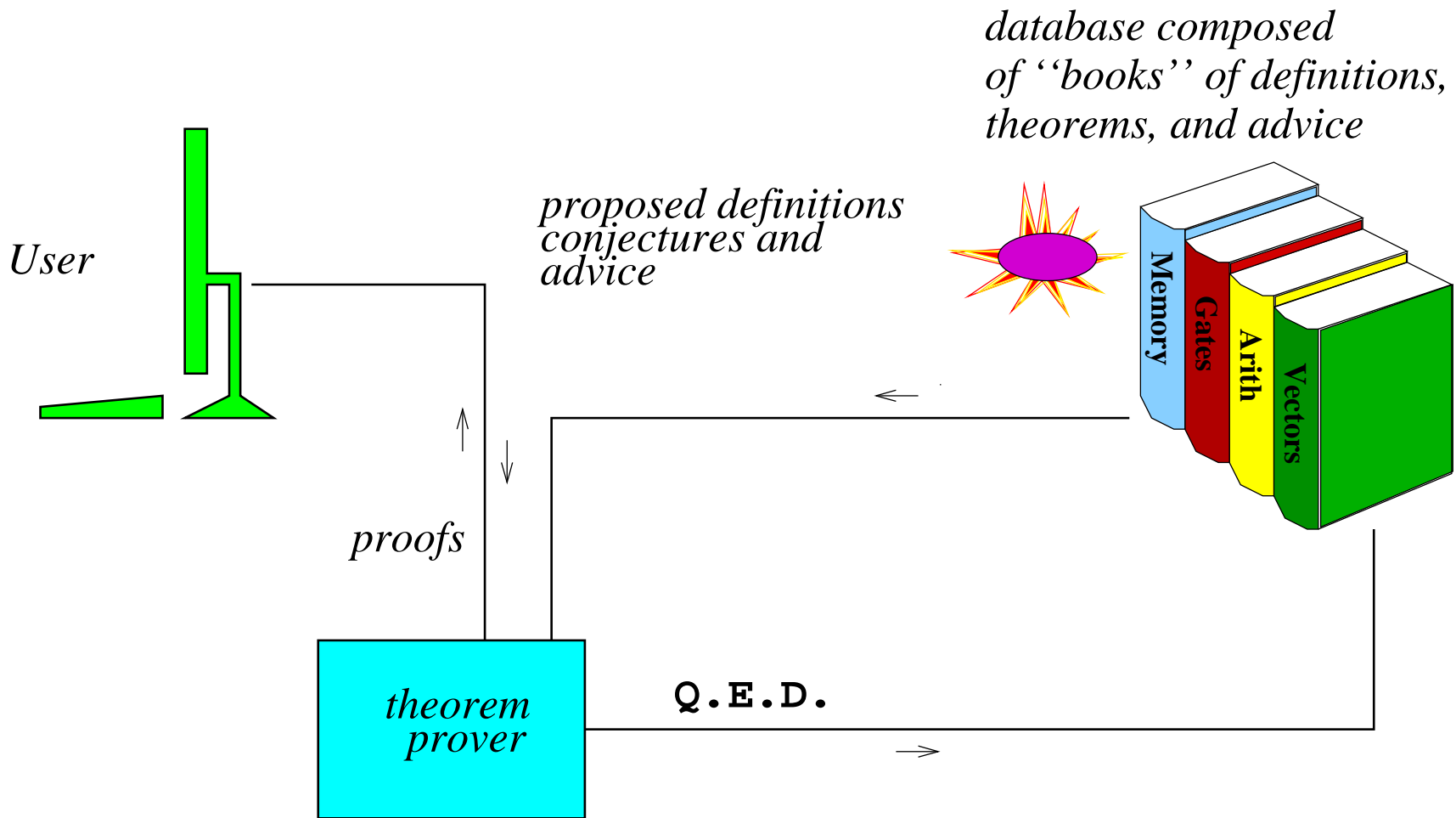










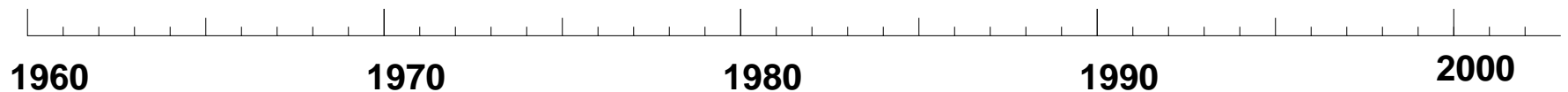


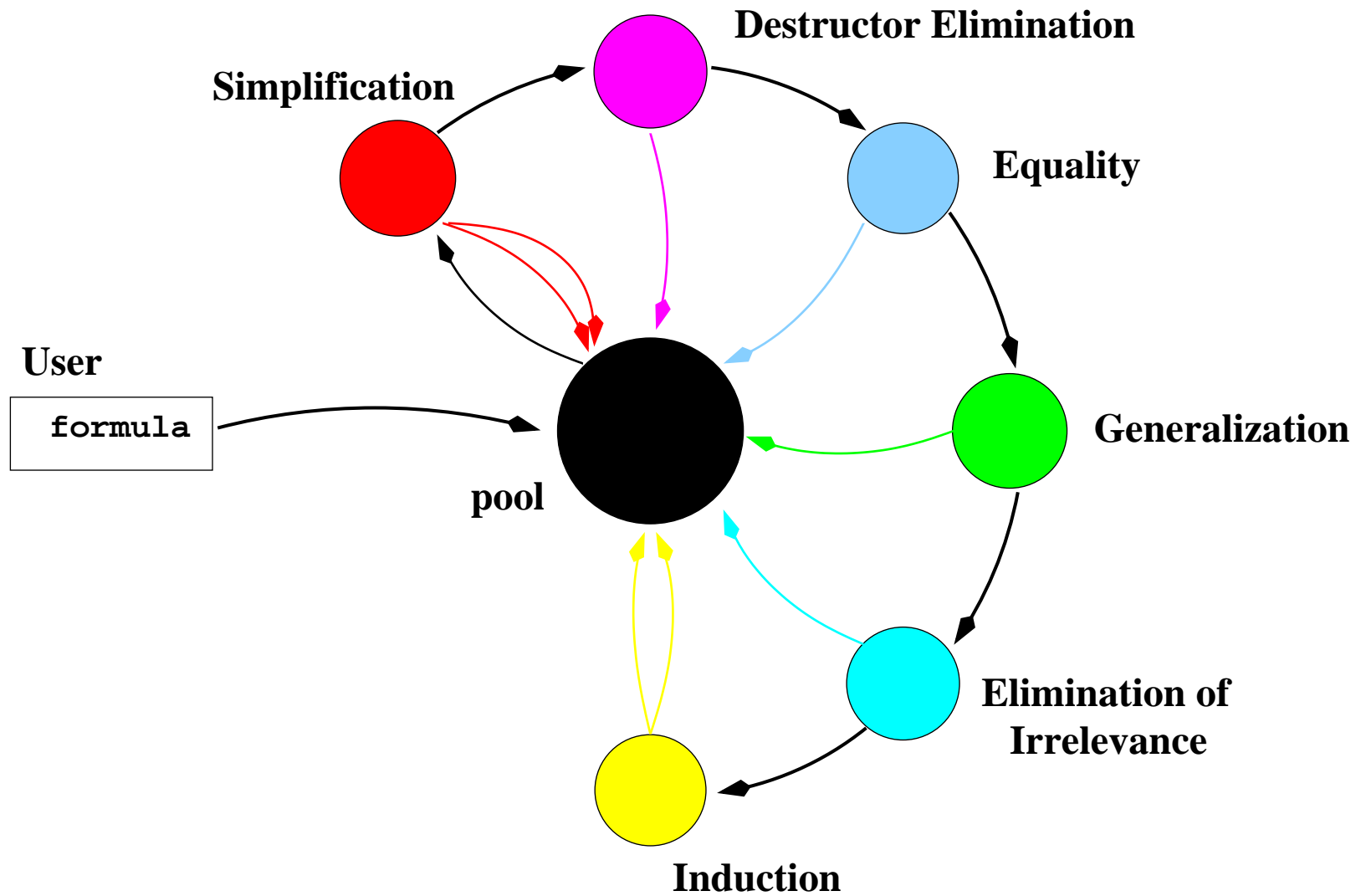
# ACL2 Demo 1

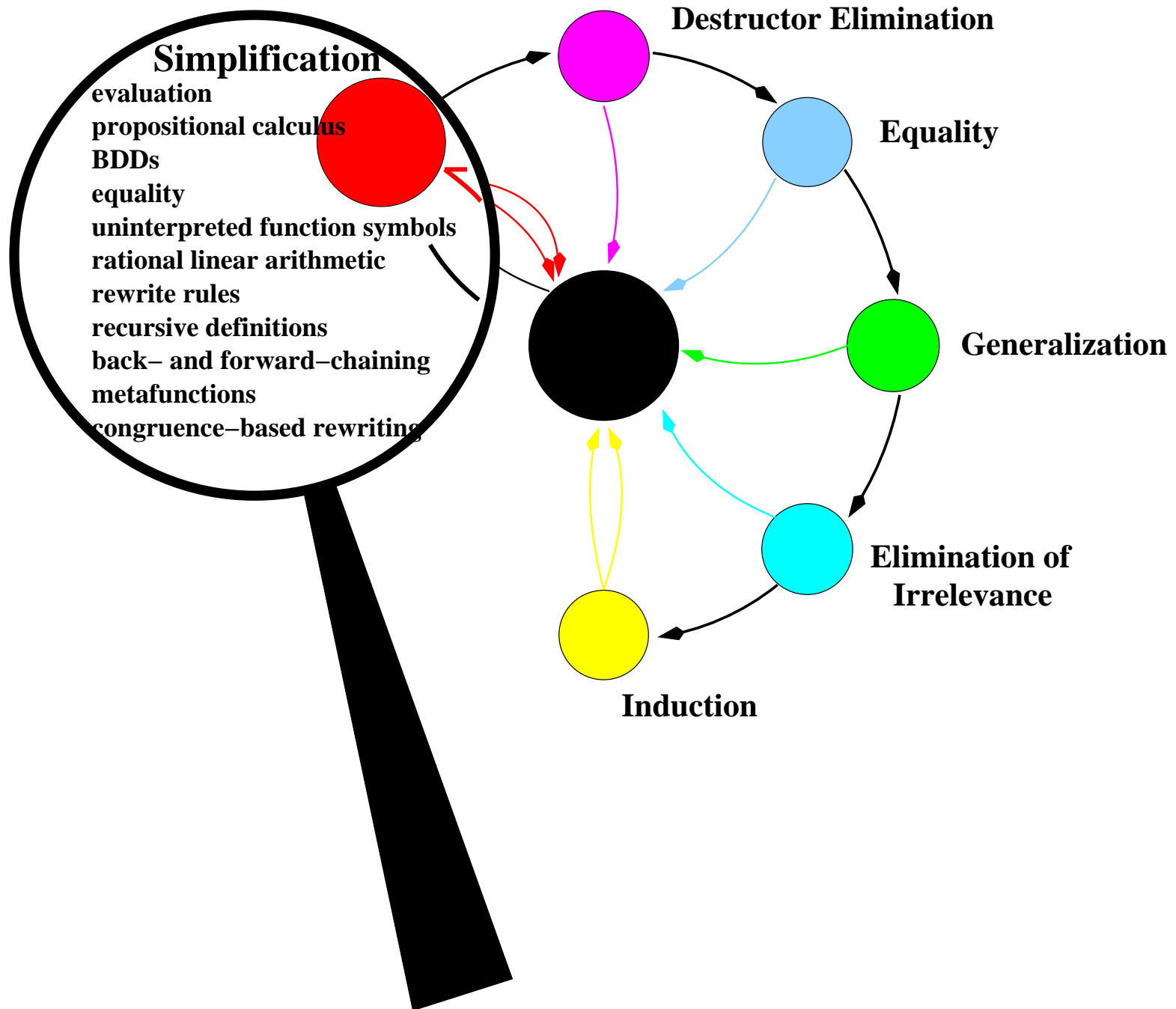
**simple list processing**

**academic math and cs**

**commercial  
applications**







# JVM Operational Semantics

M6 is an ACL2 model of the Sun JVM.

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

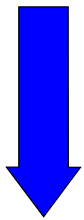
It executes most J2ME Java programs (except those with significant I/O or floating-point).

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**



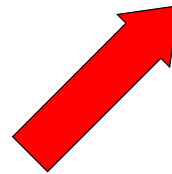
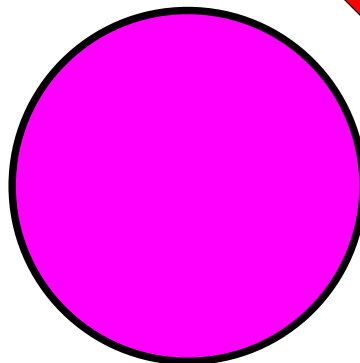
**javac**

**.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

# How To Drive ACL2

Think of ACL2 as an *assistant* in the proof discovery process.

Your role is the creative one.

ACL2's role is performing accurate, mechanical transformations using the mathematical truths you reveal to it.

# “The Method” to Prove $\alpha$

Ask ACL2 to prove  $\alpha$ .

When it fails, look at its *Key Checkpoints* and ask:

*Do I know something,  $\beta$ , that will simplify these formulas?\*\*\**

If so, use The Method to prove  $\beta$ , then start over to prove  $\alpha$ .

### **\*\*\* Major Footnote:**

Sometimes  $\beta$  takes the form of a generalization or correction to  $\alpha$ !

Sometimes it takes the form of a decomposition of the proof strategy, e.g., to prove  $\alpha_1 \rightarrow \alpha_2$  it might be best to prove  $\alpha_1 \rightarrow \gamma$  and  $\gamma \rightarrow \alpha_2$ .

# Plan

I will use The Method to prove a simple theorem.

Then I'll pose some others and we'll prove them together.

## Problem 1

Define `(splice x e y)` to replace each occurrence of `e` as an element in `y` with new elements listed in `x`.

Thus,

```
(splice '(X X) '2 '(1 2 3 4 2))
```

is

```
(1 X X 3 4 X X).
```

Prove that the number of times  $e$  occurs in  $(\text{splice } x \ e \ y)$  is the product of the number of times it occurs in  $x$  and the number of times it occurs in  $y$ .

# Solution to Problem 1

## Problem 2

Define the function `rev` to reverse a list and prove that `(rev x)` has duplicate elements precisely if `x` does.

# Solution to Problem 2

## Problem 3

Define the function to “zip” two equi-length lists together, e.g.,

```
(zipper '(a b c) '(1 2 3))
```

is

```
(a 1 b 2 c 3).
```

State and prove the distribution rule for `rev` over `zipper`.

# Solution to Problem 3

## Problem 4

State and prove the theorem that if  $x$  has duplications then there exists an element,  $e$ , of  $x$  such that  $(< 1 \text{ (how-many } e \text{ } x))$ .

$(\text{has-dups } x)$

$\rightarrow$

$(\exists e : (\text{mem } e \text{ } x) \wedge 1 < (\text{how-many } e \text{ } x))$

# Solution to Problem 4

## Recent Directions

- more efficient execution
- “or-hints” for broadening search space
- trust tags – provisions for other tools (including SAT solvers and Sixth Sense)
- proof construction and debugging tools (make-event, dmr, gag-mode)

- hash cons (Boyer and Hunt)
- execution on “generic” objects (Boyer, Hunt, Swords)
- formal verification of Boyer-Moore-like prover (Davis)

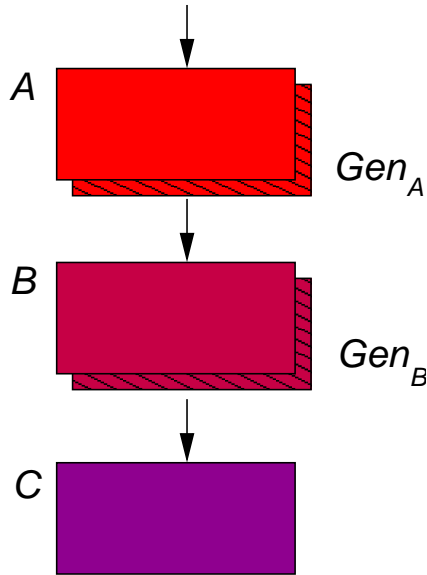
# How Do We Know ACL2 is Sound?

“Trust us!” – *Kaufmann and Moore*

Obviously, we would like to prove it correct.

But with what prover?

# “Self-Verifying Theorem Prover”



- Use  $A$  to prove  $A$  correct wrt  $B$
- Run  $Gen_A$  to get  $B$ -Level proof  $\Pi_A$
- Use  $A$  to prove  $B$  correct wrt  $C$
- Run  $Gen_B \circ Gen_A$  to get  $C$ -Level proof  $\Pi_B$
- Check  $\Pi_B$  with  $C$
- Check  $\Pi_A$  with  $B$

**Thm** (checked by  $C$ ):  $A$  is correct wrt  $C$ .

# Jared Davis' Stack "Milawa"



## **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 40 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>