An Industrially Useful Prover

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

July, 2017

Recap

Yesterday's Talk: ACL2 is used routinely in the microprocessor industry to prove theorems about designs.

Today's Talk: How ACL2 works and what it takes to use it.

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

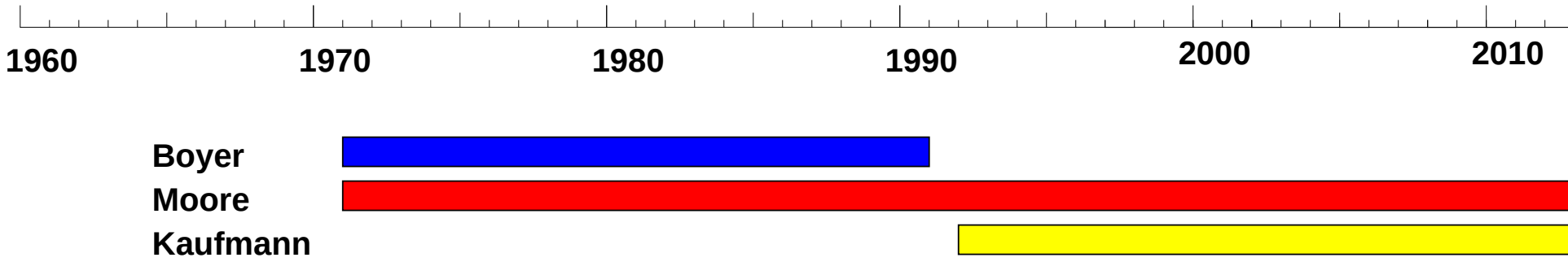
McCarthy's "Theory of Computation"

Edinburgh Pure Lisp Theorem Prover

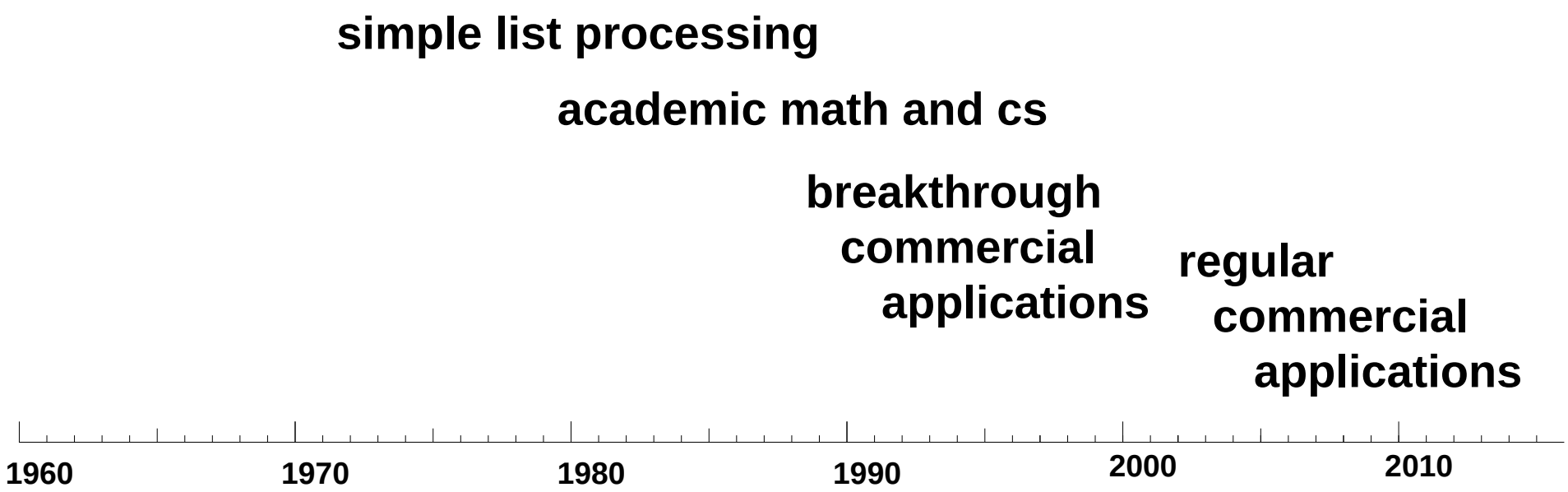
A Computational Logic

NQTHM

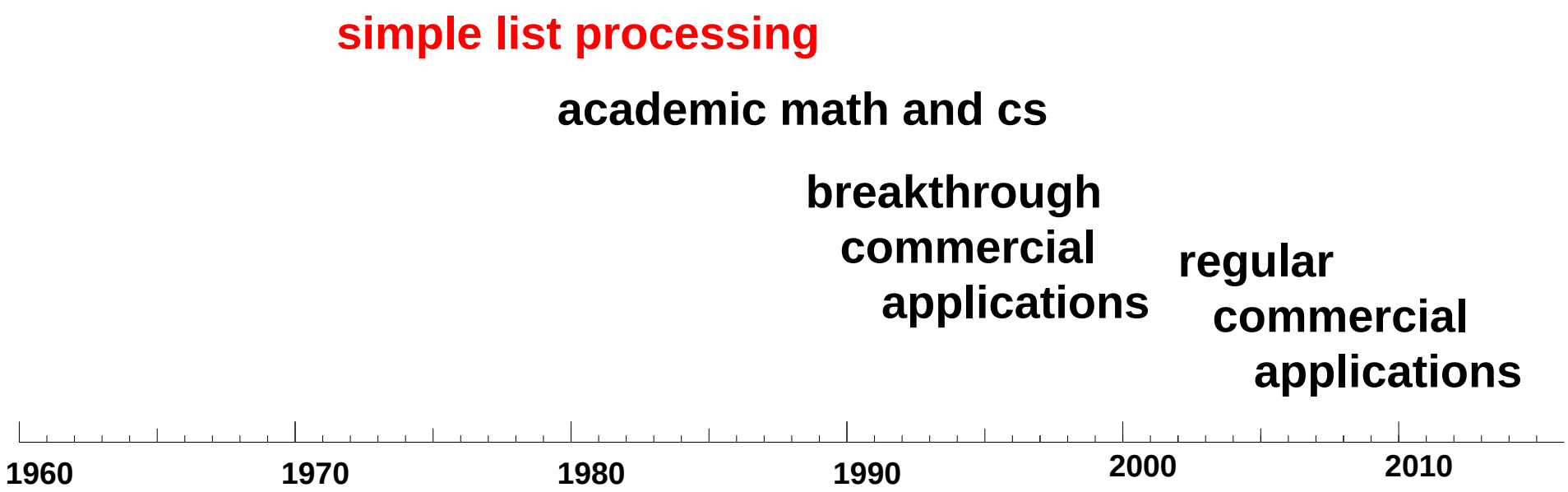
ACL2



Theorems Proved



Theorems Proved



A Few Axioms

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

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

`(cons` x y`)`: $\langle x, y \rangle$

`(car` $pair$`)` : $head(pair)$ or left component

`(cdr` $pair$`)` : $tail(pair)$ or right component

The constant `'(1 2 3)` abbreviates

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

e.g., $\langle 1, \langle 2, \langle 3, \mathbf{nil} \rangle \rangle \rangle$.

Theorems Proved: 1970s

- ap is associative:

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

$\forall a \forall b \forall c$: ap(ap(a,b),c) = ap(a,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))
= (cons 1 (ap '(2 3) '(4 5 6)))
= (cons 1 (cons 2 (ap '(3) '(4 5 6))))
= (cons 1 (cons 2 (cons 3 (ap nil '(4 5 6)))))
= '(1 2 3 4 5 6)
```

```
(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)))
```

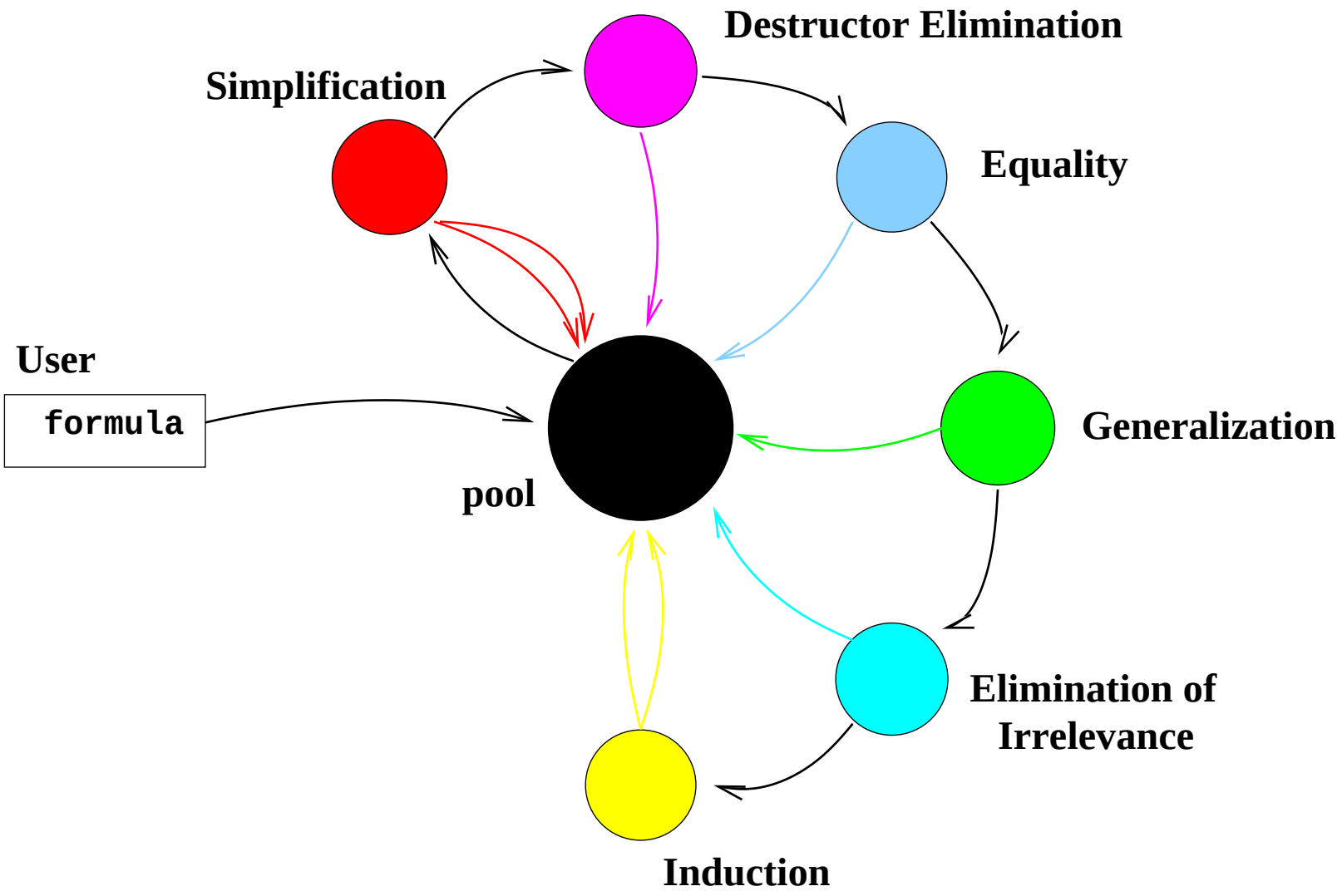
```
(equal (ap (ap a b) c)
       (ap a (ap b c)))

Proof: by induction on a.

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

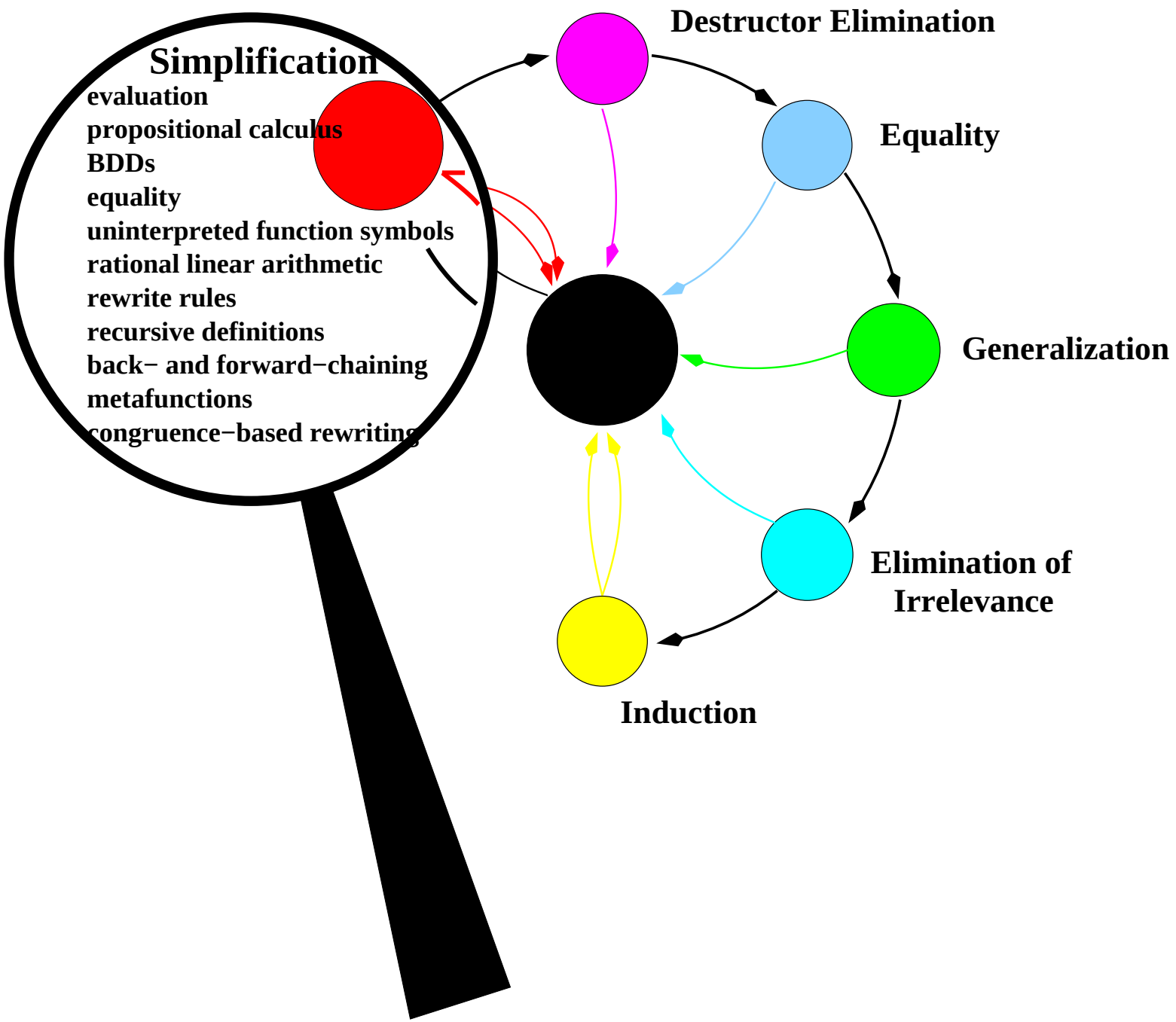
```
(equal (ap (ap a b) c)
       (ap a (ap b c)))
```

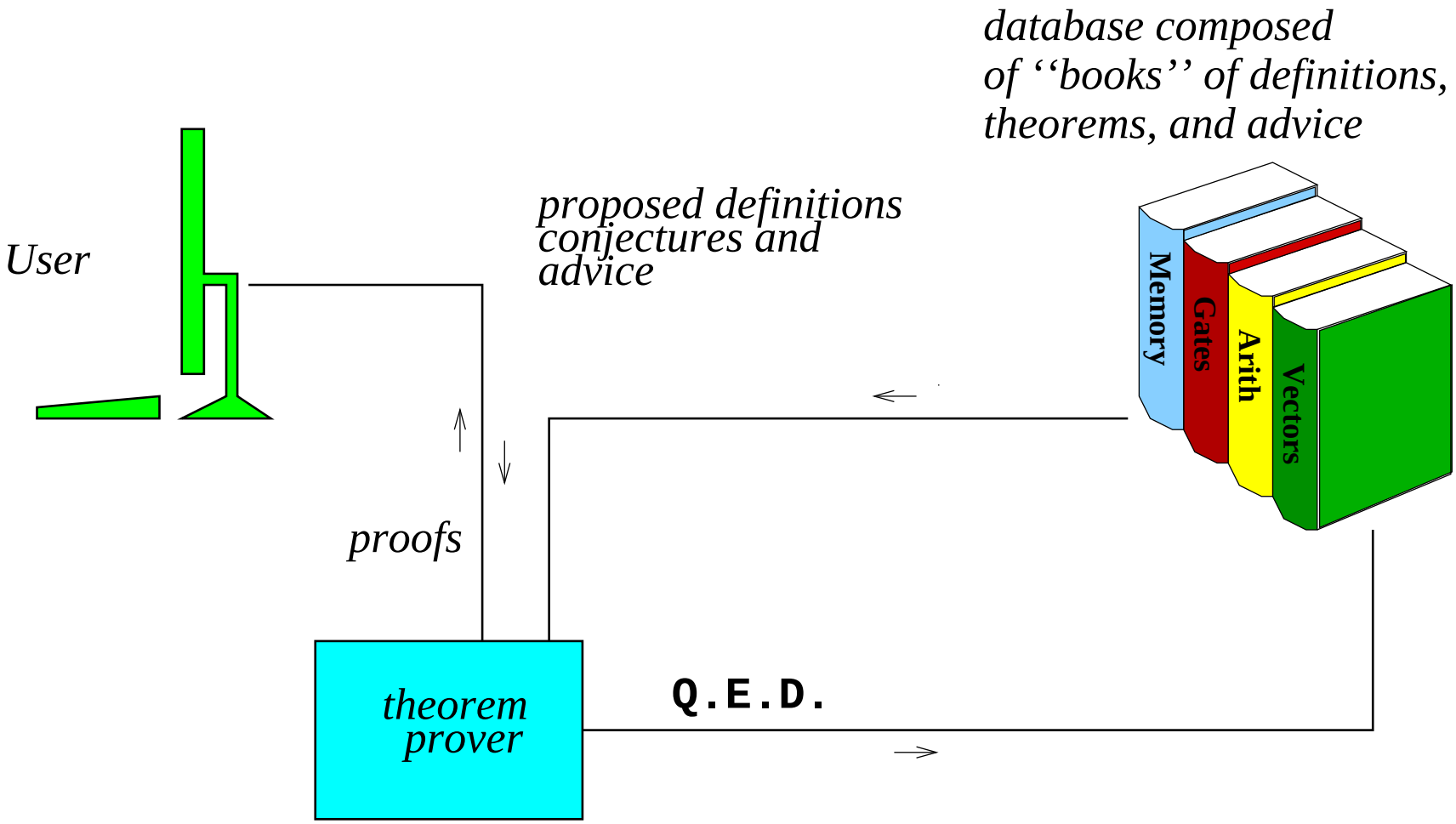
Proof: by induction on a.

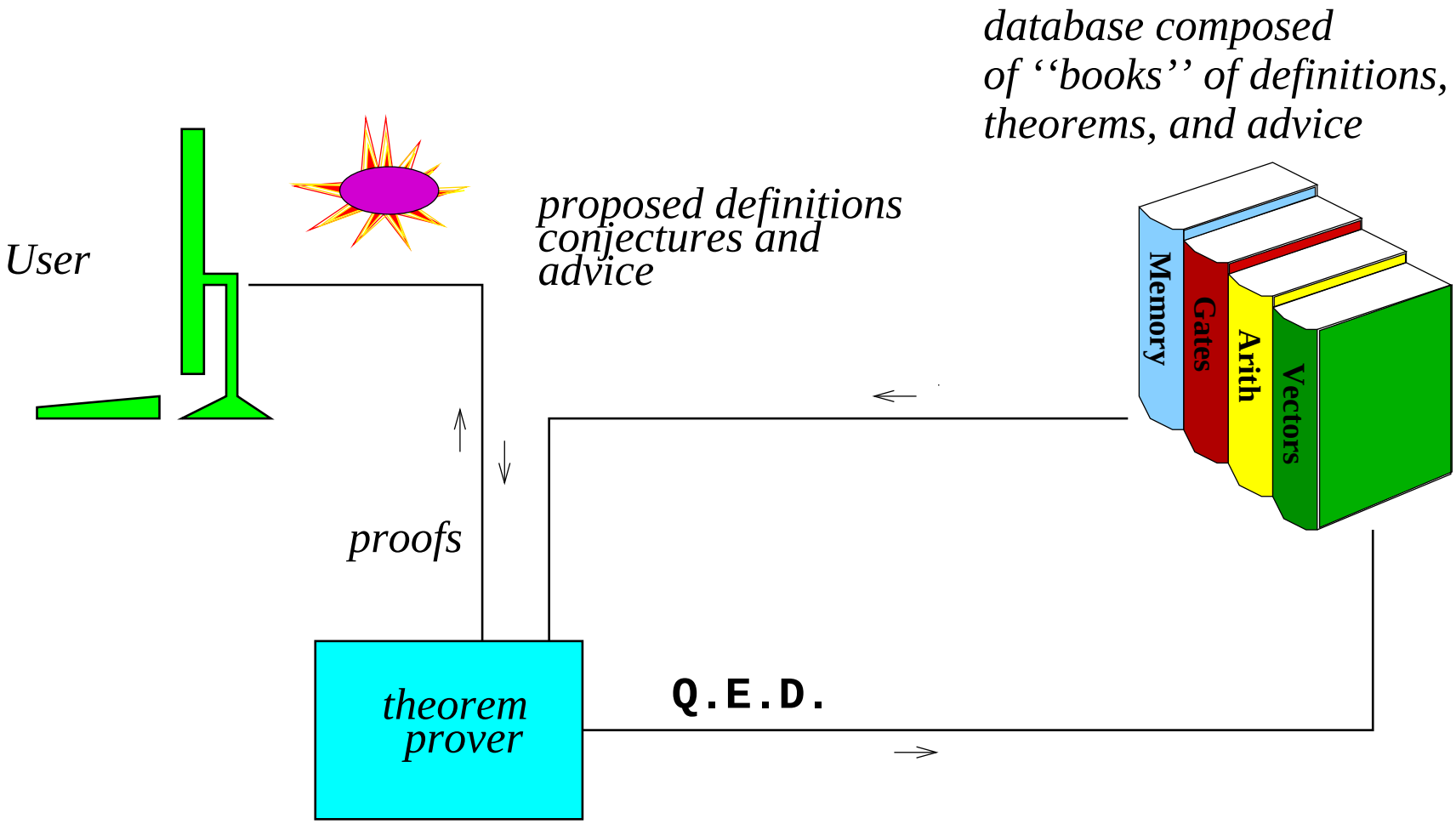
Q.E.D.

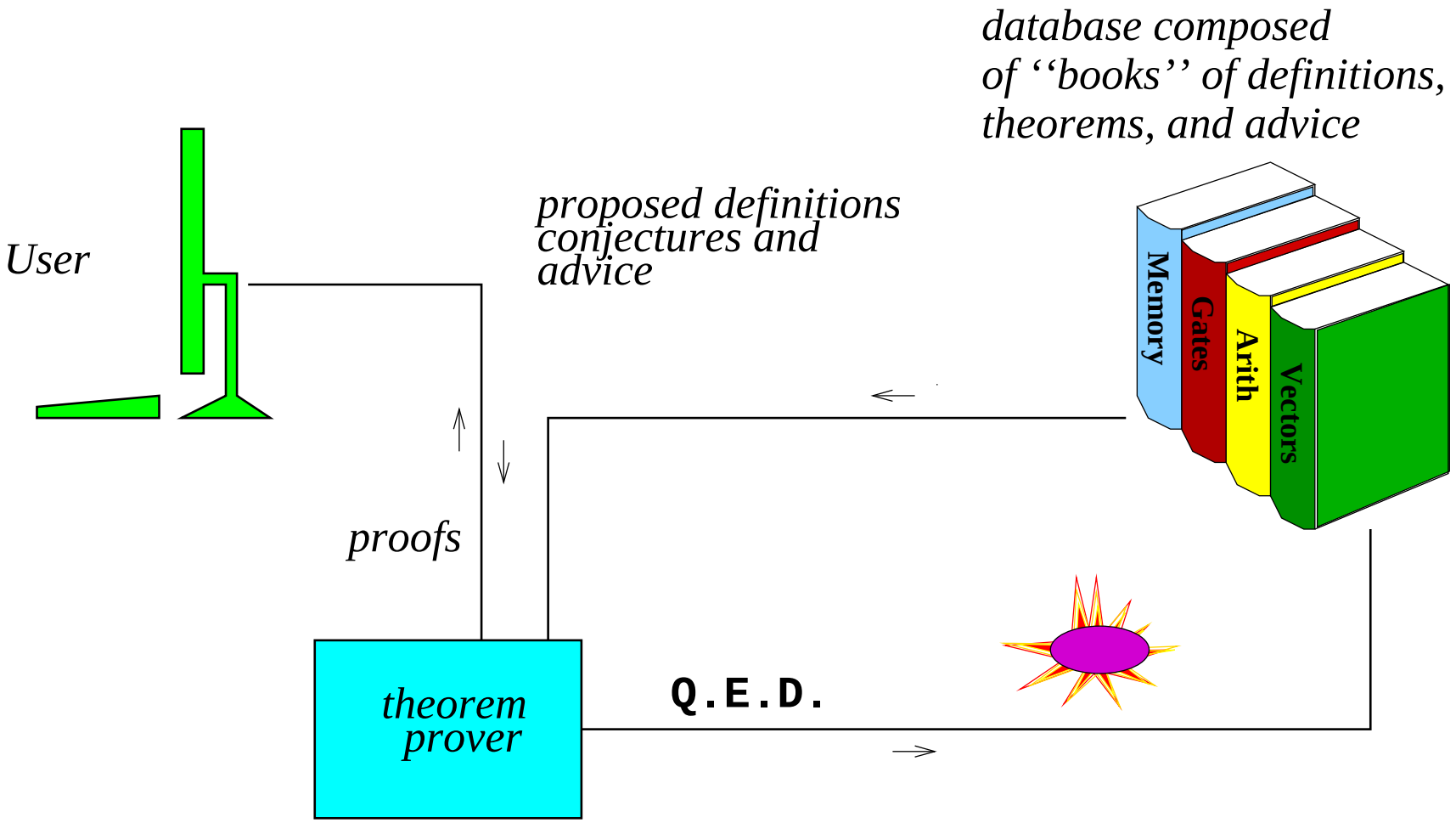
ACL2 Demo 1 – Basic

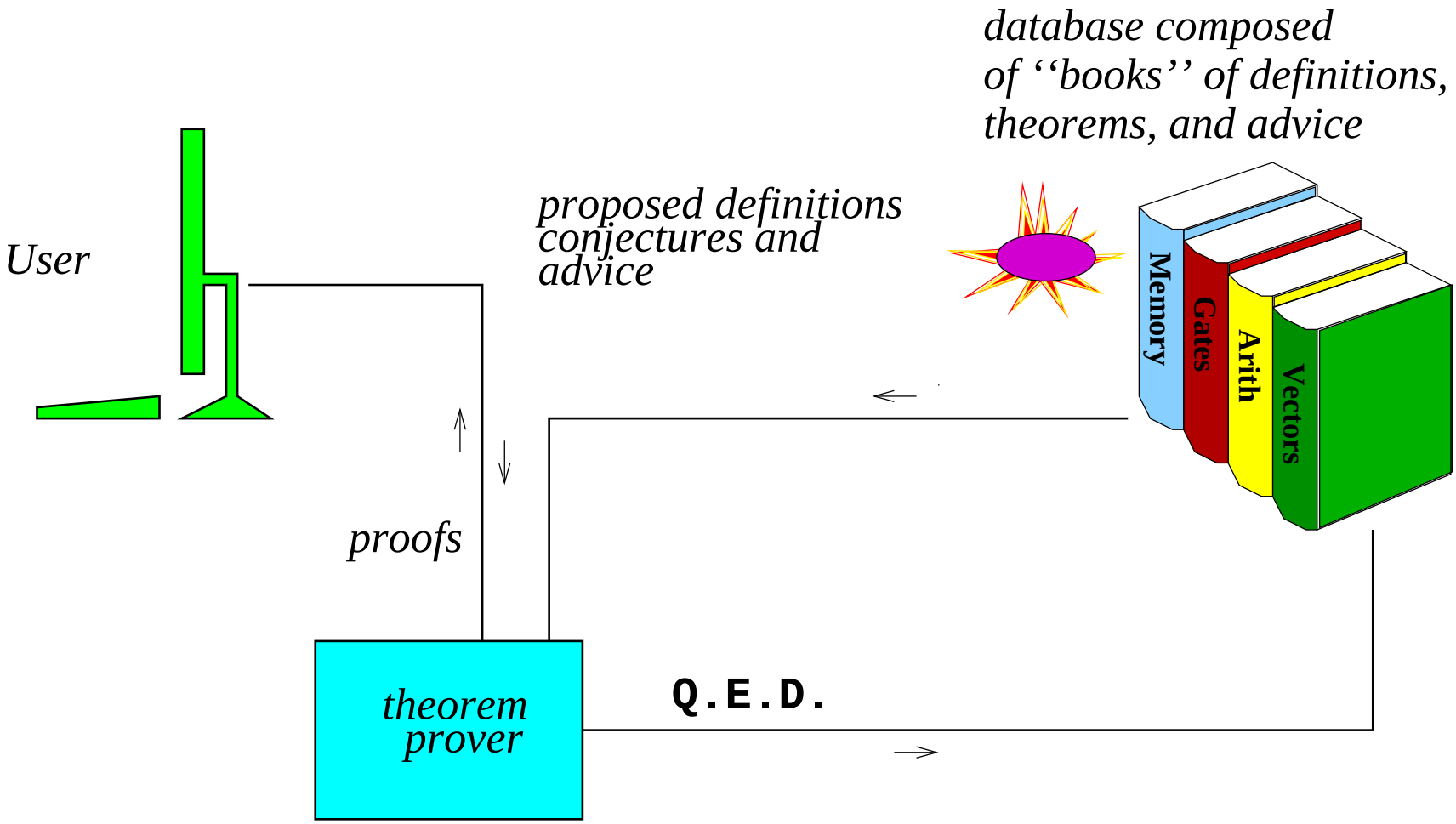

Destructor Elimination
Simplification
Equality
User
formula
Generalization
pool
Elimination of
Irrelevance
Induction


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.

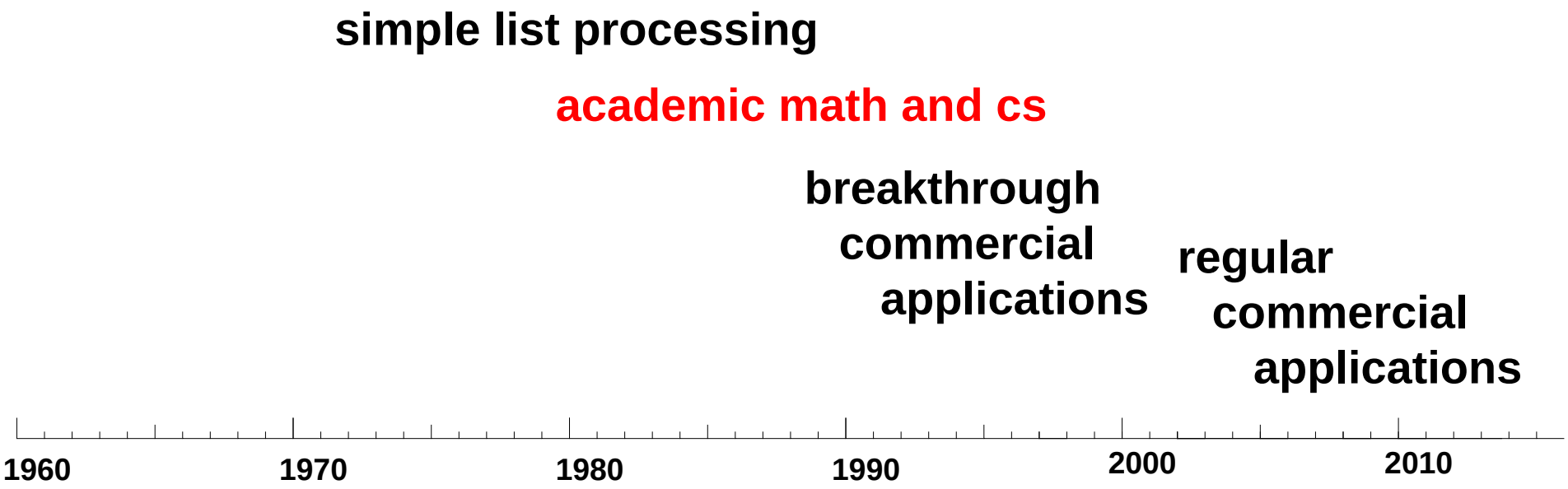

database composed
of ‘‘books’’ of definitions,
theorems, and advice
User
proposed definitions
conjectures and
advice
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.

ACL2 Demo 2 – User Guidance

ACL2 Community Books

`https://github.com/acl2/acl2/books/`
contains 5,780 user-supplied books, with 62,242 definitions and 123,804 theorems (as of Feb 2016).

Theorems Proved: 1980s



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)

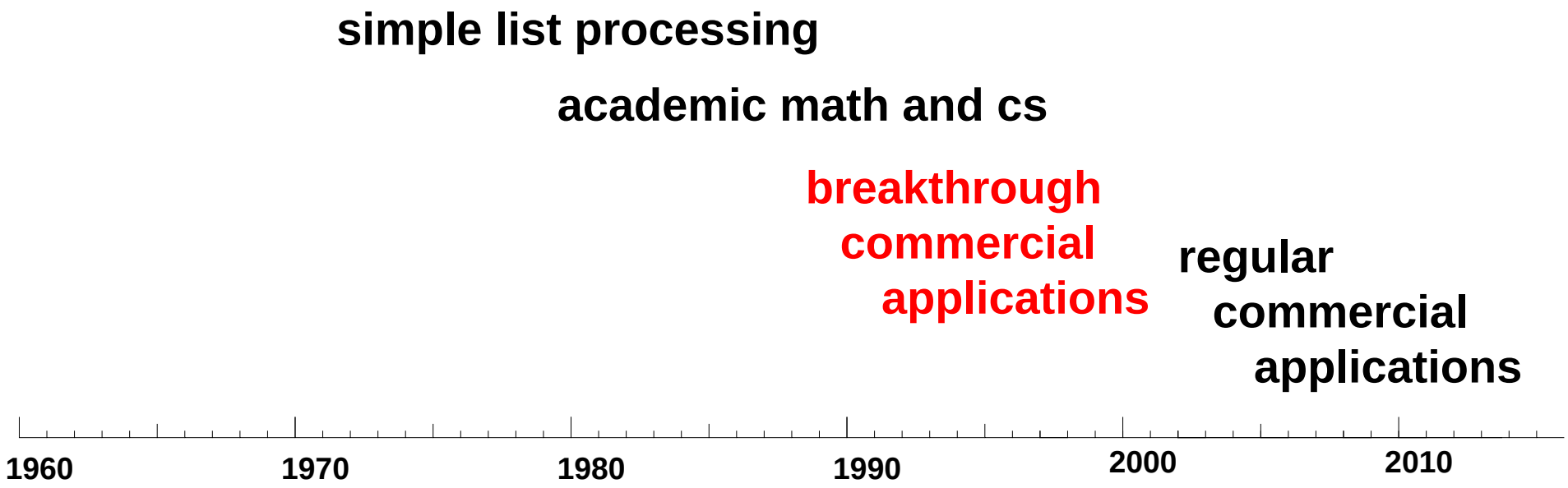
1980s Academic CS

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

All the theorems "fit together:" a theorem proved about an app holds when the binary image of the app is run at the gate level.

The gate level design was fabricated (in 1992).

Theorems Proved: 1990s



Demo 3 – Speeding Up

Verified Tools and Efficiency

ACL2 is extensible: if you program a theorem prover in ACL2 (applicative Common Lisp) and prove it correct with ACL2, then ACL2 can use your prover during its proofs.

This feature is used over 150 times in the Community Books to do things like:

- normalize arithmetic expressions denoting machine addresses
- compute bounds on some expressions
- add verified decision procedures (e.g., BDD)
- transform some formulas into different domains (e.g., bounded arithmetic problems into Boolean problems for "bit-blasting")
- checking proofs by other tools

Demo 4 – Verified Tool

Checking SAT Proofs

Marijn Heule used SAT to prove the Boolean Pythagorean Theorem: Every red/blue colouring of the positive integers contains a monochromatic solution of $a^2 + b^2 = c^2$.

This can be finitised and solved with a propositional satisfiability checker (Glucose).

The proof is big.

Proof production (solving) time:	13,516 CPU hours
Proof Size:	192 terabytes
	= 192,000 gigabytes
Proof Optimization:	22,605 CPU hours
Proof Size:	194 terabytes

Computing Platform: Lonestar5 cluster using 1200 CPUs. Total wall-clock time: 34 hours.

But SAT proofs have a history of being incorrect and so the SAT community has agreed that they should be checked by verified checkers.

SAT proof checker:

```
(implies (and (formula-p formula)
              (refutation-p proof formula))
         (not (satisfiable formula))))
```

`Refutation-p` takes about 8 pages of ACL2 code to write down.

Verified with ACL2 by Matt Kaufmann.

Proof production (solving) time:	13,516 CPU hours
Proof Size:	192 terabytes (= 192,0
Proof Optimization:	22,605 CPU hours
Proof Size:	194 terabytes
Verified ACL2 Checker:	8,651 CPU hours

The verified ACL2 checker runs at about half the speed of the fastest (unverified) checker.

But it runs about 10 times faster than the checker verified with Coq.

It is used in SAT competitions to check the claims of the winners.

Trustworthy checking of huge proofs/computations is feasible.

Take Home Message

Mechanical theorem-proving is used in industry.

Humans are needed to formalize specifications and steer the prover.

Precision and mathematical creativity are needed.

But the machine takes responsibility for the correctness of proofs.

It is both challenging and liberating.