Mechanized Operational Semantics

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

Marktoberdorf Summer School 2008

(Lecture 1: The Logic (ACL2))

Caveat

The most widely accepted meaning of *Operational Semantics* today is Plotkin's "Structural Operational Semantics" (SOS) (1981) in which the semantics is presented as a set of inference rules on syntax and "configurations" (states) defining the valid transitions.

But in these lectures I take an older approach perhaps best called *interpretive semantics* in which the semantics of a piece of code is given by a recursively defined interpreter on the syntax and a state.

I suspect the older approach came from McCarthy who wrote “*the meaning of a program is defined by its effect on the state vector*,” in “Towards a Mathematical Science of Computation” (1962).

The interpretive approach was used with mechanized support in *A Computational Logic* (Boyer and Moore, 1979) to specify and verify an expression compiler. The low level machine was defined as a recursive function on programs (sequence of instructions) against a state consisting of a push down stack and an environment assigning values to variables.

Plotkin rightly states that the interpretive approach tends to produce large and possibly unweildy states. Procedure call and non-determinism make things worse.

This is mitigated by the presence of a mechanized reasoning system. Interpretive semantics also confer certan advantages we will discuss.

The Boyer-Moore community has used *operational semantics* (in the “interpretive” sense) with great success since the mid-1970s.

So what you’re about to see is an old-fashioned but effective treatment of Operational Semantics.

End of Caveat

Outline

Lecture 1: The Logic (ACL2)

Lecture 2: An Operational Semantics

Lecture 3: Direct Code Proofs

Lecture 4: Inductive Assertion Proofs

Lecture 5: Extended Example

A Computational Logic
for
Applicative Common Lisp

- functional programming language
- mathematical logic
- mechanized theorem prover

for describing and analyzing digital systems

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

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

A C L

A C L

ACL

ACL

=

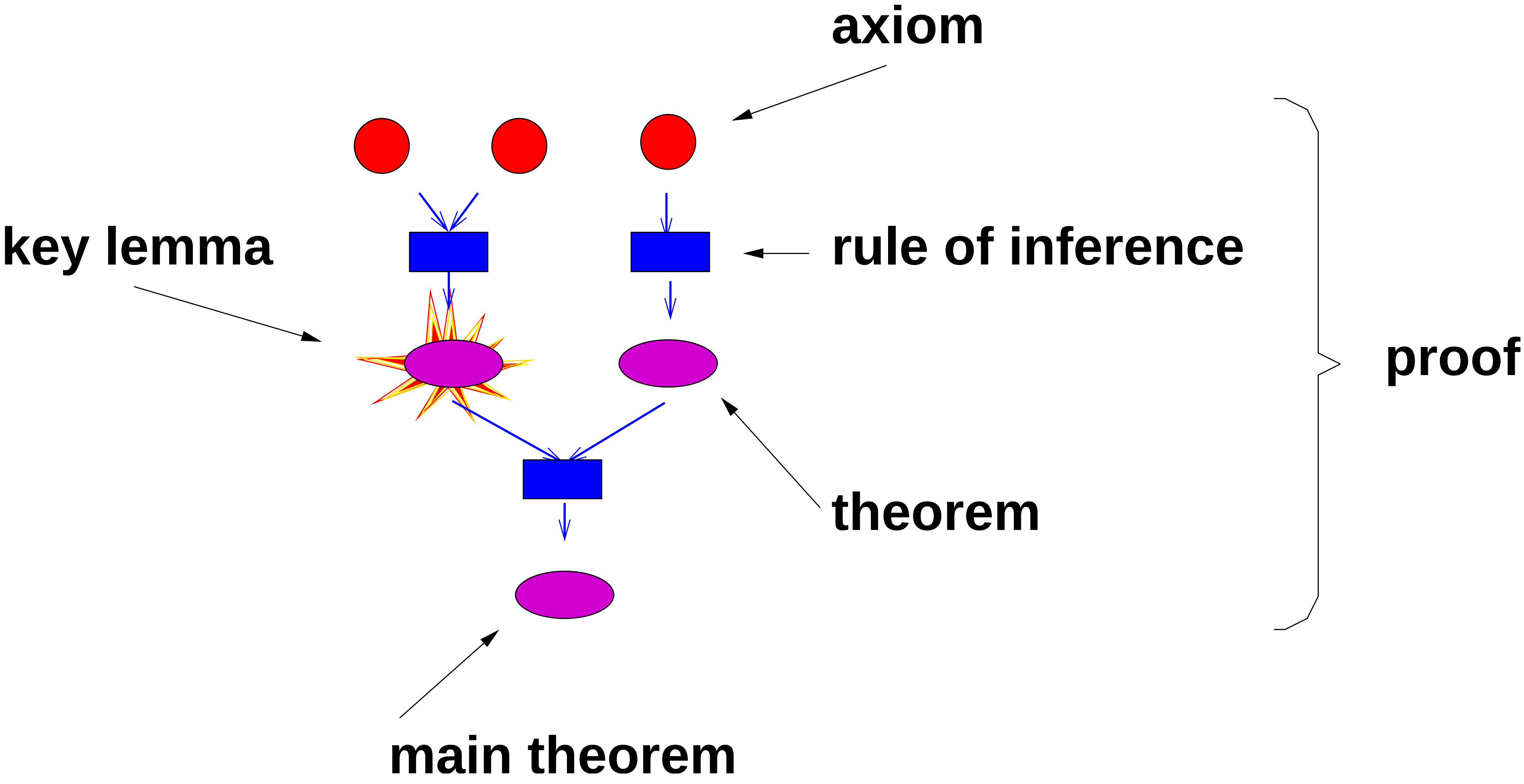
ACL2

ACL2

- functional programming language $\Leftarrow$
- mathematical logic
- mechanized theorem prover

A Formal Logic

- syntax
- axioms
- rules of inference
- semantics



axiom

key lemma

rule of inference

proof

theorem

main theorem

For Those Who Know Logic

ACL2 is a first-order, quantifier-free, untyped logic of total recursive functions.

For Those Who Know Logic

ACL2 is a first-order[1], quantifier-free[2], untyped[3] logic of total[4] recursive functions.

[1] But see `functional-instantiation`.

[2] But see `defchoose`.

[3] But see `guard`.

[4] But see `defpun`.

Example Terms

ACL2 term	*traditional notation*
`(sqrt (log 2 i))`	$\mathrm{sqrt}(\log(2, i))$
	$\sqrt{\log_2 i}$
`(+ x (* 3 (expt y 2)))`	$x + 3 \times y^2$
`(cons (car x) rest)`	$\mathrm{cons}(\mathrm{car}(x), rest)$

Whitespace Is Ok

```
(firstn (length (terminal-substring j dt)) pat
```

Whitespace Is Ok

```
(firstn (length (terminal-substring j dt))
        pat)
```

Whitespace Is Ok

```
(firstn (length
            (terminal-substring j dt))
         pat)
```

Whitespace Is Ok

```
(firstn (length
          (terminal-substring
            j
            dt))
        pat)
```

Whitespace Is Ok

```
(firstn
 (length
  (terminal-substring
   j
   dt))
 pat)
```

Data Types

ACL2 supports five disjoint data types:

- numbers
- characters
- strings
- symbols
- pairs

About T and NIL

`T` and `NIL` are used as the "truth values" true and false.

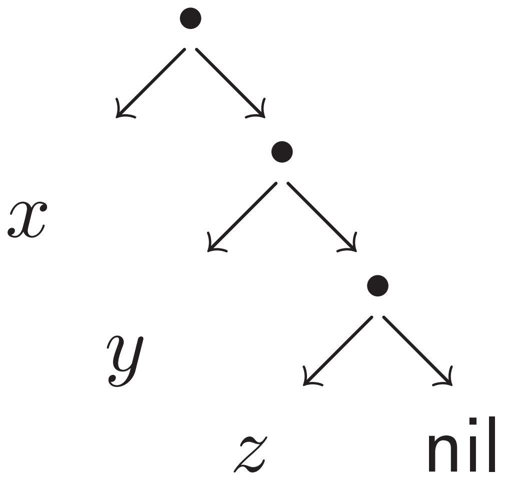
`NIL` is *also* used as the "terminal marker" on nested pairs representing lists. (More later.)

Informally, "`NIL` is the empty list."

But `T` and `NIL` are *symbols*!

About Pairs

$< x, < y, < z, \texttt{nil} >>>$



$(x \;\; y \;\; z)$

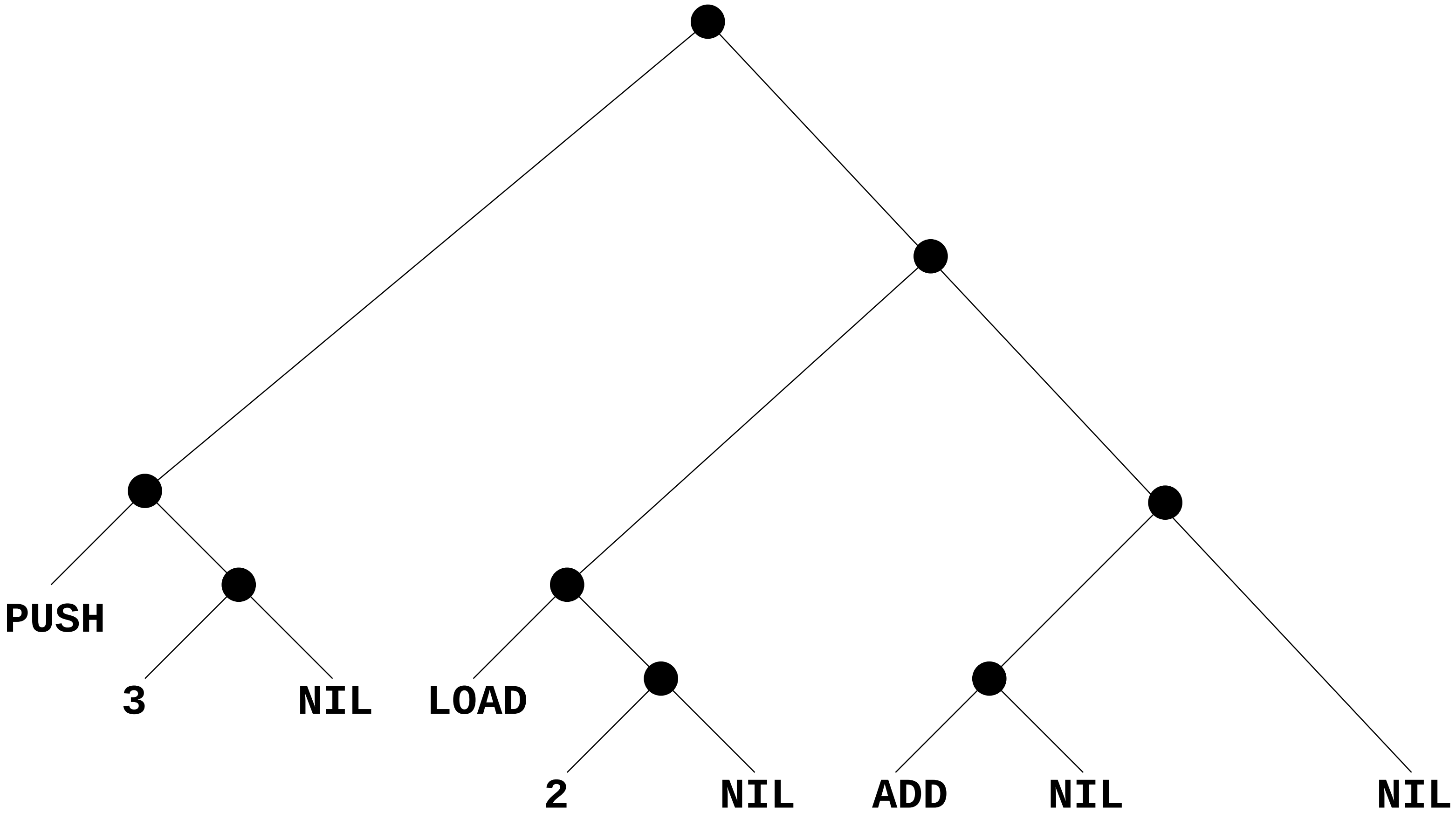
Atoms

An *atom* is any ACL2 object other than a pair.

So here are some atoms: `123`, `nil`, `COLOR`.

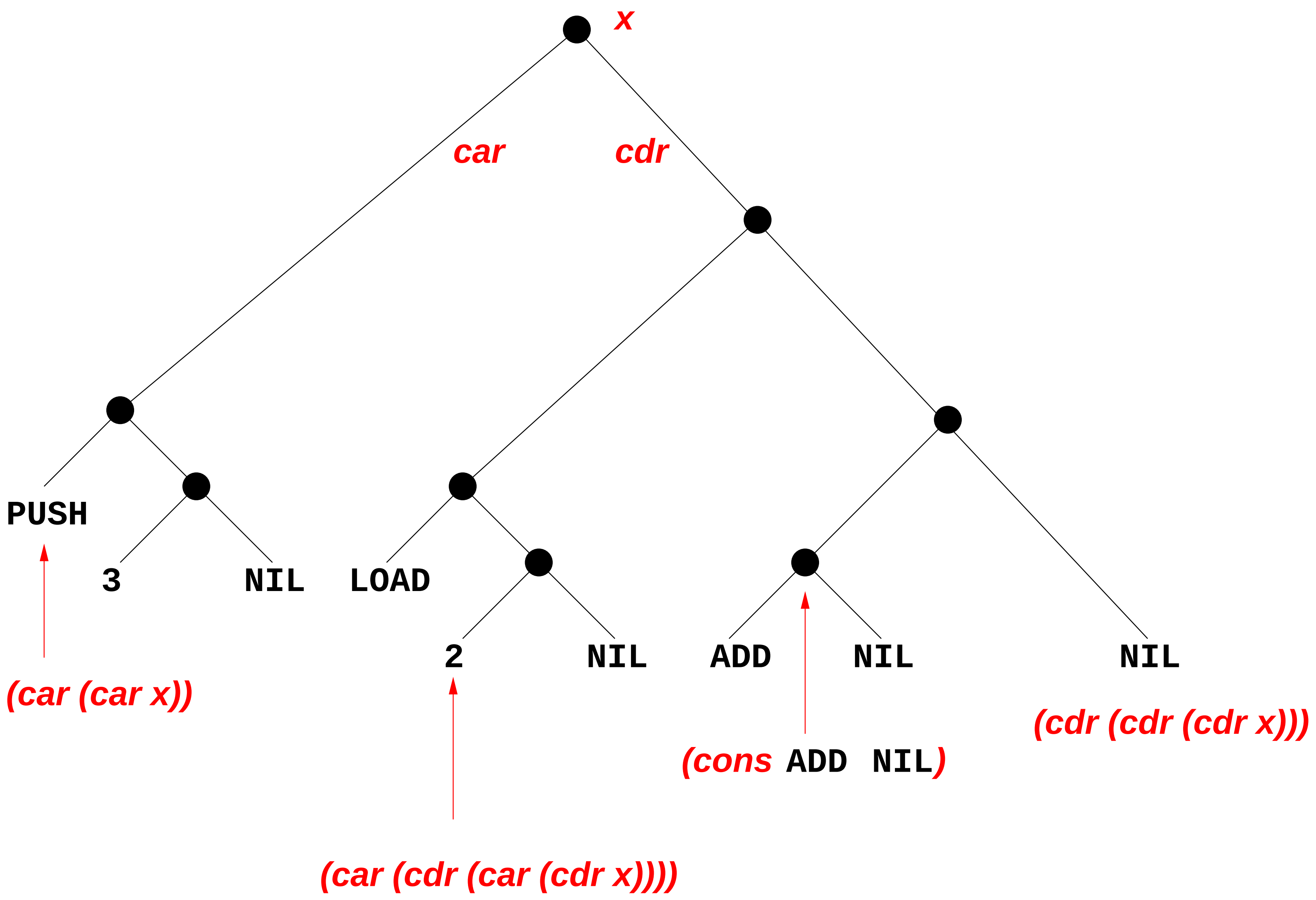
Here is a non-atom: `(PUSH 3)`

`((PUSH 3) (LOAD 2) (ADD))`



Primitive Functions

- `(cons` x y`)` – the ordered pair $\langle x, y\rangle$
- `(car` x`)` – left component of x, if x is a pair; else `nil`
- `(cdr` x`)` – right component of x, if x is a pair; else `nil`
- `(consp` x`)` – `t` if x is a pair; else `nil`


x
car
cdr
PUSH
3
NIL
LOAD
2
NIL
ADD
NIL
NIL
(car (car x))
(cons ADD NIL)
(cdr (cdr (cdr x)))
(car (cdr (car (cdr x))))

Axioms

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

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

$\texttt{(consp x)} = \texttt{t} \vee \texttt{(consp x)} = \texttt{nil}$

$\texttt{(consp (cons x y))} = \texttt{t}$

$\texttt{(consp x)} = \texttt{nil} \rightarrow \texttt{(car x)} = \texttt{nil}$

$$\texttt{(consp x)} = \texttt{nil} \rightarrow \texttt{(cdr x)} = \texttt{nil}$$

$$\texttt{(consp x)} = \texttt{t} \rightarrow \texttt{(cons (car x) (cdr x))} = \texttt{x}$$

$$\texttt{(symbolp x)} = \texttt{t} \rightarrow \texttt{(consp x)} = \texttt{nil}$$

$$\texttt{(integerp x)} = \texttt{t} \rightarrow \texttt{(consp x)} = \texttt{nil}$$

Primitive Functions (Continued)

- `(equal` x y`)` – `t` if x is y; else `nil`
- `(if` x y z`)` – if x is `t` then y; else z (non-Boolean x are treated as `t`)
- `(+` x y`)` – sum of x and y (non-numbers are treated as 0)

- `(-` x y`)` – difference of x and y (non-numbers are treated as 0)

- `(*` x y`)` – product of x and y (non-numbers are treated as 0)

- `(zp` x`)` – `t` if x is 0; else `nil` (non-*naturals* are treated as 0!)

Defining Functions

```
(defun endp (x) (not (consp x)))

(defun atom (x) (not (consp x)))

(defun not (p) (if p nil t))

(defun and (p q) (if p q nil))

(defun or (p q) (if p p q))
```

```
(defun implies (p q)
  (if p (if q t nil) t))

(defun iff (p q)
  (and (implies p q) (implies q p)))

(defun natp (x)
   (and (integerp x)
        (<= 0 x)))
```

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$0 \prec 1 \prec 2 \prec \ldots$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$0 \prec 1 \prec 2 \prec \ldots \prec \omega$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$$0 \prec 1 \prec 2 \prec \ldots \prec \omega \prec \omega + 1$$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$$0 \prec 1 \prec 2 \prec \ldots \prec \omega \prec \omega{+}1 \prec \omega{+}2 \prec \ldots$$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$$0 \prec 1 \prec 2 \prec \ldots \prec \omega \prec \omega{+}1 \prec \omega{+}2 \prec \ldots$$

$$\ldots \prec \omega \times 2$$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$$0 \prec 1 \prec 2 \prec \ldots \prec \omega \prec \omega{+}1 \prec \omega{+}2 \prec \ldots$$

$$\ldots \prec \omega \times 2 \prec \omega \times 2 + 1 \prec \ldots$$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$0 \prec 1 \prec 2 \prec \ldots \prec \omega \prec \omega{+}1 \prec \omega{+}2 \prec \ldots$

$\ldots \prec \omega \times 2 \prec \omega \times 2 + 1 \prec \ldots$

$\ldots \prec \omega^2$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$$0 \prec 1 \prec 2 \prec \ldots \prec \omega \prec \omega{+}1 \prec \omega{+}2 \prec \ldots$$

$$\ldots \prec \omega \times 2 \prec \omega \times 2 + 1 \prec \ldots$$

$$\ldots \prec \omega^2 \prec \ldots \prec \omega^3 \prec \ldots$$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$$0 \prec 1 \prec 2 \prec \ldots \prec \omega \prec \omega+1 \prec \omega+2 \prec \ldots$$

$$\ldots \prec \omega \times 2 \prec \omega \times 2 + 1 \prec \omega \times 2 + 2 \ldots$$

$$\ldots \prec \omega^2 \prec \ldots \prec \omega^3 \prec \ldots$$

$$\ldots \prec \omega^\omega$$

The Ordinals

The ordinals are a well-ordered extension of the natural numbers.

$$0 \prec 1 \prec 2 \prec \ldots \prec \omega \prec \omega{+}1 \prec \omega{+}2 \prec \ldots$$

$$\ldots \prec \omega \times 2 \prec \omega \times 2 + 1 \prec \omega \times 2 + 2 \ldots$$

$$\ldots \prec \omega^2 \prec \ldots \prec \omega^3 \prec \ldots$$

$$\ldots \prec \omega^\omega \prec \ldots \prec \omega^{\omega^{\omega^{\omega^{\cdots}}}} = \epsilon_0$$

Ordinals below ϵ_0 can be represented with lists (Cantor's canonical form).

For example,

$$\omega^{\omega+3} \times 27 + \omega^{100} + \omega^3 \times 238 + \omega \times 3 + 798$$

is represented by

```
((((1 . 1) . 3) . 27) (100 . 1) (3
. 238) (1 . 3) . 798)
```

Ordinals below ϵ_0 can be represented with lists (Cantor's canonical form).

The recognizer for such ordinals can be defined recursively.

The "less than" relation, $\prec$, can be defined recursively.

Primitive Functions (continued)

- `(o-p` x`)` – `t` if x represents an ordinal below ϵ_0; else `nil`
- `(o<` x y`)` – the well-founded ordering $\prec$ on ordinals below ϵ_0

Induction and Recursion

Recursive definitions are admissible only if some measure of the arguments can be proved to decrease in a well-founded ordering, typically some ordinal measure ordered by o<.

Inductions are justified by a well-founded ordering. Given a measure and ordering, you can assume any “smaller” instance of the conjecture being proved.

Induction and recursion are duals.

```
(defun len (x)
  (if (endp x)
      0
    (+ 1 (len (cdr x)))))
```

`(len '(a b c))` $\Rightarrow$ 3

('$\Rightarrow$' means "evaluates to (reduces under the axioms to the constant)".)

```
(defun len (x)
  (if (endp x)
      0
    (+ 1 (len (cdr x)))))
```

Why is this admissible?

```
(defun len (x)
  (if (endp x)
      0
    (+ 1 (len (cdr x)))))
```

Theorem:

$$\neg\ endp(x) \ \rightarrow\ size(cdr(x)) \ \prec\ size(x)$$

```
(defun len (x)
  (if (endp x)
      0
    (+ 1 (len (cdr x)))))
```

Theorem:

```
(implies (not (endp x))
         (o< (size (cdr x))
             (size x)))
```

Induction (suggested by `(len x)`)

To prove $\psi(x, y)$ it is sufficient to prove:

Base Case:

`(implies (endp x)` $\psi(x, y)$`)`

Induction Step:

```
(implies (and (not (endp x))
              $\psi$((cdr x), $\alpha$))
         $\psi(x,y)$)
```

Every total recursive function suggests an induction.

We won't discuss it further, but that is key to the automation of induction.

```
(defun nth (n x)
  (if (zp n)
      (car x)
    (nth (- n 1) (cdr x))))
```

(nth 3 ’(A B C D E)) ⇒ D.

```
(defun char (s n)
  (nth n (coerce s ’list)))
```

`(char "Hello" 1)` ⇒ `#\e`
(the lowercase character ‘e’).

```
(defun update-nth (n v x)
  (if (zp n)
      (cons v (cdr x))
    (cons (car x)
          (update-nth (- n 1) v (cdr x)))))
```

```
(update-nth 3 'X '(A B C D E))
```
$\Rightarrow$ `(A B C X E)`.

```
(defun member (e x)
  (if (endp x)
      nil
    (if (equal e (car x))
        x
      (member e (cdr x)))))
```

`(member 3 '(1 2 3 4 5))` ⇒ `(3 4 5)`.

```
(defun repeat (x n)
  (if (zp n)
      nil
    (cons x (repeat x (- n 1)))))
```

`(repeat t 4)` ⇒ `(t t t t)`

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

`(append '(A B C) '(D E))`
⇒ `(A B C D E)`.

```
(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)).

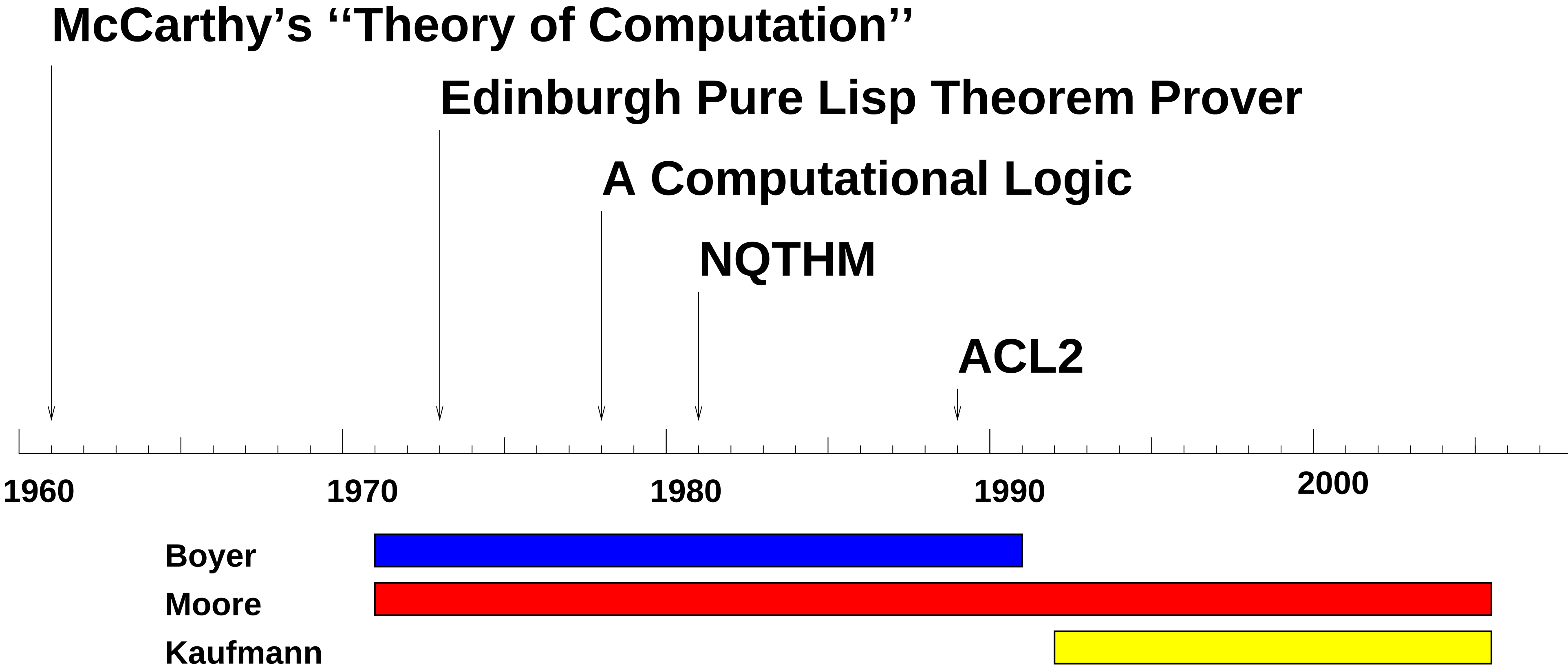
T

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

Proof: by induction on a.

Q.E.D.

Boyer-Moore Project



McCarthy’s ‘‘Theory of Computation’’

Edinburgh Pure Lisp Theorem Prover

A Computational Logic

NQTHM

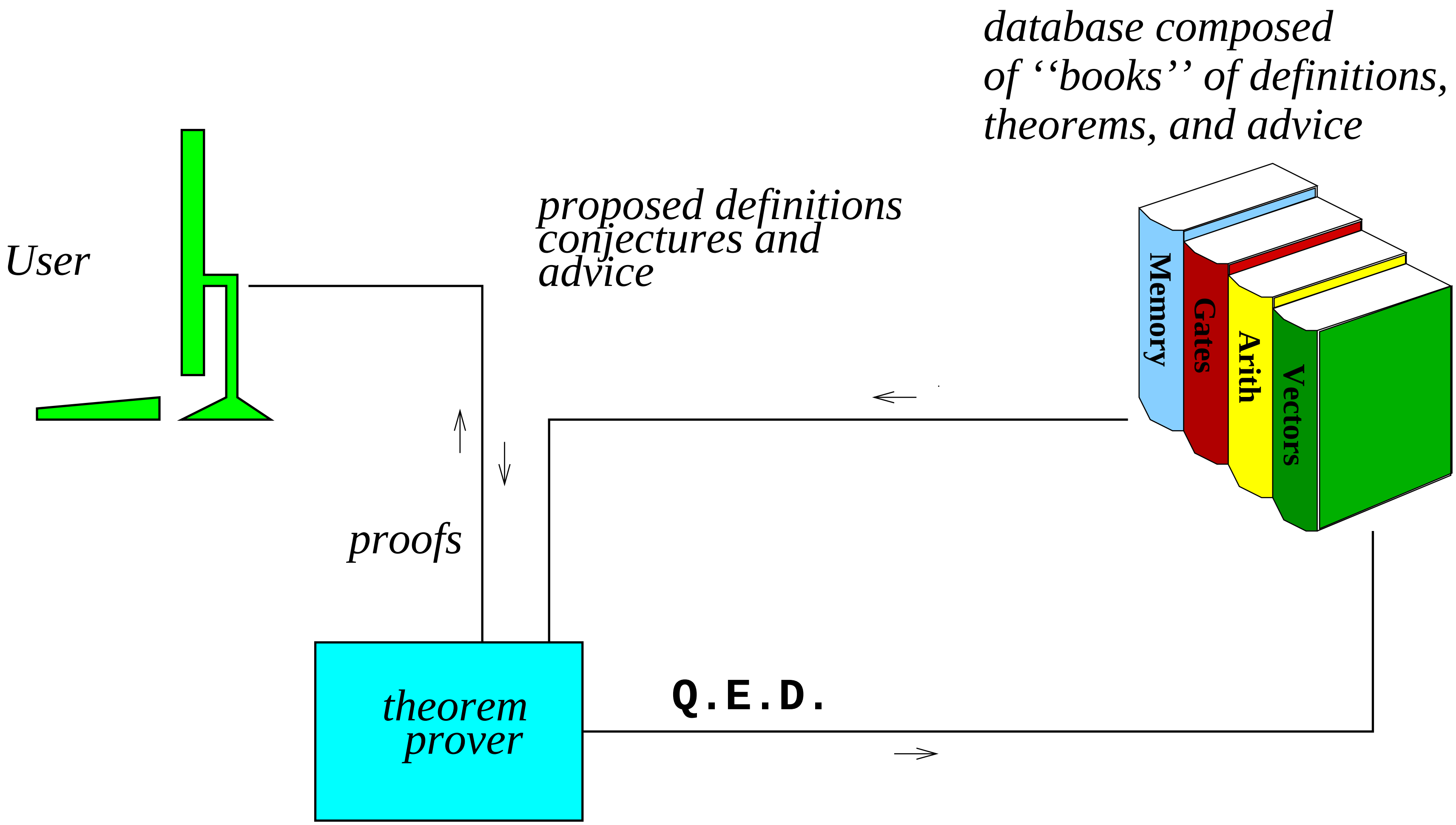
ACL2

1960 1970 1980 1990 2000

Boyer

Moore

Kaufmann



database composed
of ‘‘books’’ of definitions,
theorems, and advice

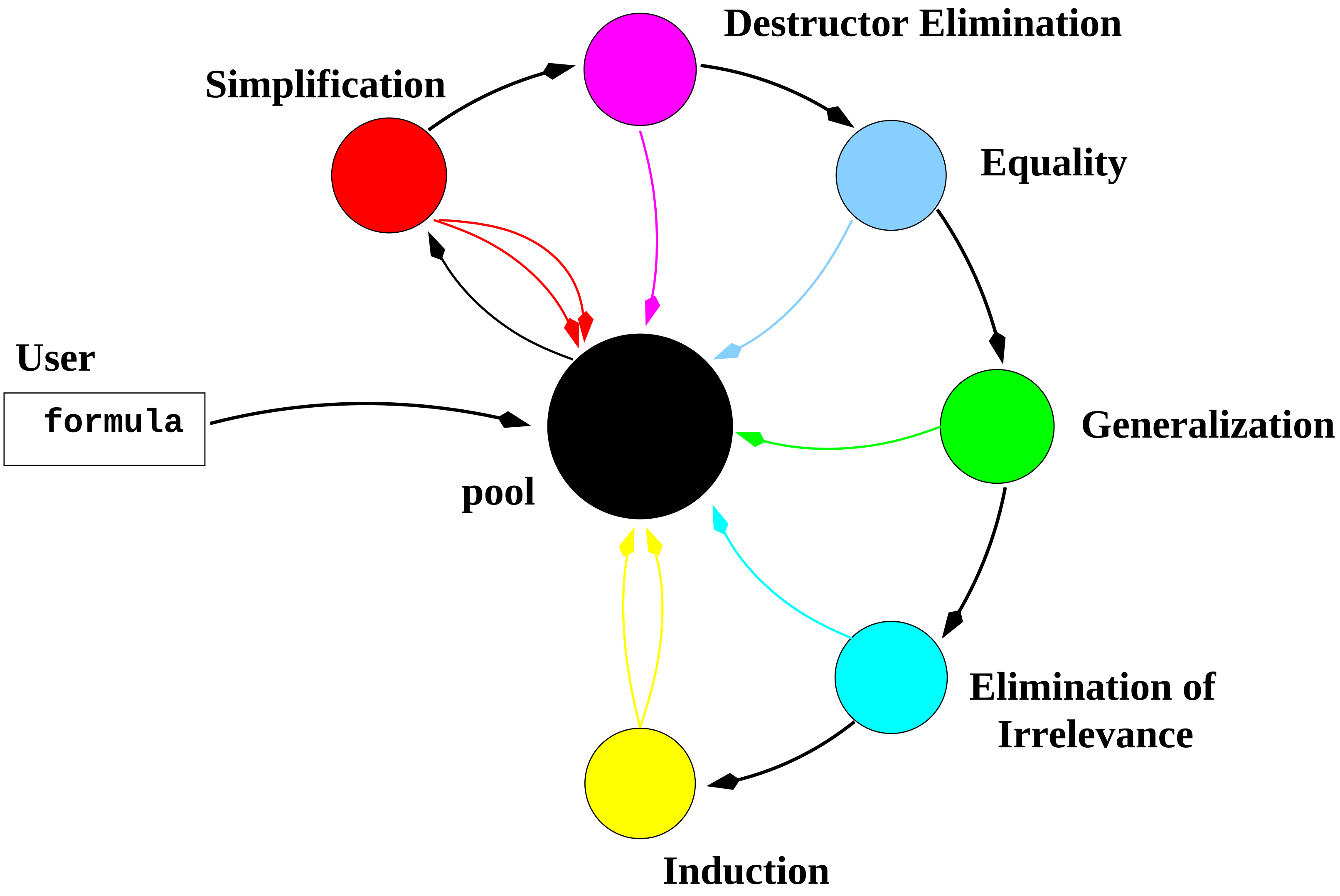
User

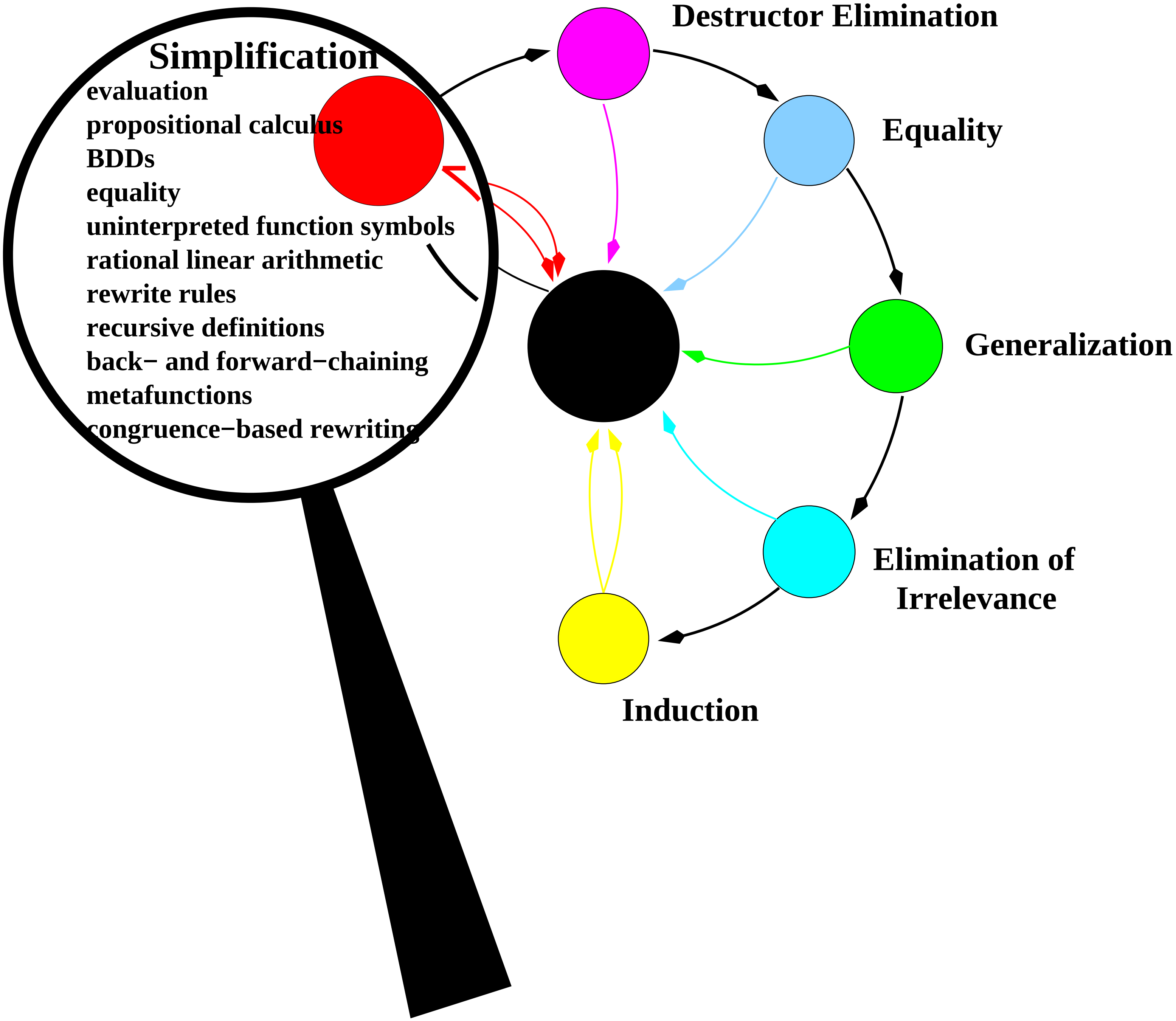
proposed definitions
conjectures and
advice

proofs

theorem
prover

Q.E.D.


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

ACL2 Demo 1

Books

The ACL2 user develops *books* that tailor the system to find proofs in a given domain.

The user provides *proof sketches* in the form of sequences of key lemmas.

The system fills in the gaps.

This enables *proof maintenance*.

Minor modifications to previously proved theorems (or previously analyzed formal models) can often be verified without user intervention – because the books encode a *strategy* not a *proof*.

Next Time

An operational semantics for a simple language.