Little (but Hard) Theorems About Big Systems: Some Case Studies

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

Marktoberdorf Summer School 2004

Lecture 2

A Clarification

- `(1 2 3)` is the object containing (successively) the elements 1, 2, and 3.
- `'(1 2 3)` is a *term* whose value is the object `(1 2 3)`.

Every Lisp term is a Lisp object.

Not every Lisp object is a Lisp term.

Recursive Definitions

```
(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
      y))

(app '(1 2 3) '(4 5 6))
```

$\Rightarrow$

???

Recursive Definitions

```
(defun app (x y);x='(1 2 3) y='(4 5 6)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
      y))

(app '(1 2 3) '(4 5 6))
⇒
???
```

Recursive Definitions

```
(defun app (x y);x='(1 2 3) y='(4 5 6)
  (if t
      (cons (car x)
            (app (cdr x) y))
    y))

(app '(1 2 3) '(4 5 6))
⇒
???
```

Recursive Definitions

```
(defun app (x y);x='(1 2 3) y='(4 5 6)

      (cons (car x)
            (app (cdr x) y))
      )

(app '(1 2 3) '(4 5 6))
⇒
???
```

Recursive Definitions

```
(defun app (x y);x='(1 2 3) y='(4 5 6)

     (cons '1
          (app (cdr x) y))
     )

(app '(1 2 3) '(4 5 6))
⇒
???
```

Recursive Definitions

```
(defun app (x y);x='(1 2 3) y='(4 5 6)

      (cons 1
            (app (cdr x) y))
      )

(app '(1 2 3) '(4 5 6))
⇒
???
```

Recursive Definitions

```
(defun app (x y);x='(1 2 3) y='(4 5 6)

      (cons 1
            (app '(2 3) y))
      )

(app '(1 2 3) '(4 5 6))
⇒
???
```

Recursive Definitions

```
(defun app (x y);x='(1 2 3) y='(4 5 6)

      (cons 1
            (app '(2 3) y))
      )

(app '(1 2 3) '(4 5 6))
```

$\Rightarrow$

???

Recursive Definitions

```
(defun app (x y);x='(1 2 3) y='(4 5 6)

      (cons 1
            (app '(2 3) y))
      )

(cons 1 (app '(2 3) '(4 5 6)))
```

$\Rightarrow$

```
???
```

Recursive Definitions

```
(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
      y))

(cons 1 (app '(2 3) '(4 5 6)))
```

$\Rightarrow$

???

Recursive Definitions

```
(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
      y))
```

```
(cons 1 (cons 2 (app '(3) '(4 5 6))))
```

$\Rightarrow$

???

Recursive Definitions

```
(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
      y))
```

```
(cons 1 (cons 2 (cons 3 (app nil '(4 5 6)
```

$\Rightarrow$

???

Recursive Definitions

```
(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
      y))

(cons 1 (cons 2 (cons 3 '(4 5 6))))
```

$\Rightarrow$

???

Recursive Definitions

```
(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
     y))
```

```
(cons 1 (cons 2 (cons 3 '(4 5 6))))
```

$\Rightarrow$

```
(1 2 3 4 5 6)
```

The ACL2 Formal System

- syntax
- axioms
- rules of inference (prop calc and equality)
- definitional principle
- induction principle

A Few “Axioms”

1 $\texttt{t} \neq \texttt{nil}$

2 $\texttt{x} \neq \texttt{nil} \rightarrow \texttt{(if x y z)} = \texttt{y}$

3 $\texttt{x} = \texttt{nil} \rightarrow \texttt{(if x y z)} = \texttt{z}$

4 $\texttt{(equal x y)} = \texttt{nil} \vee \texttt{(equal x y)} = \texttt{t}$

5 $\texttt{x} = \texttt{y} \leftrightarrow \texttt{(equal x y)} = \texttt{t}$

6 (consp x) $=$ nil $\vee$ (consp x) $=$ t

7 (consp (cons x y)) $=$ t

8 (consp nil) $=$ nil

9 (car (cons x y)) $=$ x

10 (cdr (cons x y)) $=$ y

11 (consp x)$=$t

$\rightarrow$(cons (car x) (cdr x)) $=$ x

Recall

```
(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
      y))
```

Theorem?

```
(equal (app (app a b) c)
       (app a (app b c)))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: By induction on a.

Base Case:

```
(not (consp a))
→
(equal (app (app a b) c)
       (app a (app b c)))
```

Base Case

```
(not (consp a))
→
(equal (app (if (consp a)
                (cons (car a)
                      (app (cdr a) b))
                b) c)
       (app a (app b c)))
```

Base Case:

```
(not (consp a))
→
(equal (app (if (consp a)
                (cons (car a)
                      (app (cdr a) b))
                b) c)
       (app a (app b c)))
```

```
Base Case:
(not (consp a))
→
(equal (app (if nil
                (cons (car a)
                      (app (cdr a) b))
                b) c)
       (app a (app b c)))
```

```
Base Case:
(not (consp a))
→
(equal (app (if nil
                (cons (car a)
                     (app (cdr a) b))
                b) c)
       (app a (app b c)))
```

```
Base Case:
(not (consp a))
→
(equal (app b c)
       (app a (app b c)))
```

```
Base Case:
(not (consp a))
→
(equal (app b c)
       (app a (app b c)))
```

```
Base Case:
(not (consp a))
→
(equal (app b c)
       (app b c))
```

```
Base Case:
(not (consp a))
→
(equal (app b c)
       (app b c))
```

```
Base Case:
(not (consp a))
→
T
```

```
Induction Step:
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal (app (app a b) c)
       (app a (app b c)))
```

Induction Step:

```
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal (app (cons (car a)
                  (app (cdr a) b)) c)
       (app a (app b c)))
```

Induction Step:

```
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal (app (cons (car a)
                  (app (cdr a) b)) c)
       (app a (app b c)))
```

```
Induction Step:
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal (cons (car a)
         (app (app (cdr a) b) c))
       (app a (app b c)))
```

```
Induction Step:
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal (cons (car a)
         (app (app (cdr a) b) c))
       (app a (app b c)))
```

Induction Step:

```
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal (cons (car a)
          (app (app (cdr a) b) c))
       (cons (car a)
          (app (cdr a) (app b c))))
```

Induction Step:

```
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))

→
(equal (cons (car a)
         (app (app (cdr a) b) c))
       (cons (car a)
         (app (cdr a) (app b c))))
```

Induction Step:

```
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal
         (app (app (cdr a) b) c)

         (app (cdr a) (app b c)))
```

Induction Step:

```
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
```

```
Induction Step:
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
```

```
Induction Step:
(consp a) ∧
(equal (app (app (cdr a) b) c)
       (app (cdr a) (app b c)))
→
T
```

Q.E.D.

A Demonstration of ACL2

```
(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
      y))
```

Induction and recursion are duals.

Theorem

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Simplification

Simplification is the repeated replacement of equals by equals to put a term into some normal form.

If “feels” like *symbolic computation*.

Test

Can you prove

```
(equal (app (app a a) a)
       (app a (app a a)))
```

directly by induction?

Answer: NO!

```
(implies
 (and
  (consp a)
  (equal (app (app (cdr a) (cdr a)) (cdr a))
         (app (cdr a) (app (cdr a) (cdr a)))))
 (equal (app (app a a) a)
        (app a (app a a))))
```

Answer: NO!

```
(implies
 (and
  (consp a)
  (equal (app (app (cdr a) (cdr a)) (cdr a))
         (app (cdr a) (app (cdr a) (cdr a)))))
 (equal (app (app (cdr a) a) a)
        (app (cdr a) (app a a))))
```

You *must* generalize the conjecture to prove it!

Demo of ACL2

Simplify, Induct, and Simplify

Try to keep your proofs in the “simplify, induct, and simplify” style.

When the final simplification doesn’t reduce the goal to `t`, *think*!

Homework

Problem 2.0 If you haven't already, read "How to Prove Theorems Formally!"

The paper gives 26 hints for how to manage proofs (most of which have nothing to do with ACL2).

Do the exercises in the paper!

Prove two facts about subp, where

```
(defun subp (x y);‘‘subset’’ for lists
  (if (consp x)
      (if (memp (car x) y)
          (subp (cdr x) y)
        nil)
    t))

(subp ’(a b a) ’(a b c)) ⇒ t
(subp ’(a b d) ’(a b c)) ⇒ nil
```

Problem 2.1. Reflexivity

```
(subp x x)
```

Problem 2.2. Transitivity

```
(implies (and (subp x y)
              (subp y z))
         (subp x z))
```

Don’t be put off by the Lisp notation!

`Subp` and `memp` are about as basic as you can get: primitive recursive functions on sequences.

Typing (or its absence) is no problem here.

You can define analogues in any useful programming language.

You write programs *many times more complicated* every day!

Reflexivity and transitivity are among the most basic of properties.

You believe your programs have vastly more complicated properties.

Every computer scientist ought to be able to prove `subp` is reflexive and transitive!