Proving Programs by Structural Induction

Robert Keller

March 2009

Verification Patterns

- **Structural induction** *on* ***program structure*** involves proving properties based on the manner in which programs are composed (e.g. Hoare logic).

- **Transition induction** involves proving properties base on the number of ***steps executed*** (e.g. Floyd assertions).

- **Structural induction** ***on data*** involves proving properties based on the definition of data types involved.

Structural Induction on Data

- This is the "original" structural induction.
- It is expressed most easily for functional programs.
- However, we can transform any (sequential) program into a functional program (using McCarthy's transformation).
- So little generality, if any, is lost.

Structural Induction on Data

- The idea is a generalization of mathematical induction:

  If our only data type were the natural numbers, we know how to prove properties P for all such numbers:

$$\frac{P(0) \qquad\qquad P(n) \rightarrow P(n+1)}{(\forall n)\, P(n)}$$

Structural Induction on Lists

- $$\frac{P(\,[\;]\,) \qquad\qquad P(L) \rightarrow P(\,[A \mid L]\,)}{(\forall L)\,P(L)}$$
- Here
  - [ ] is the empty list
  - L represents an arbitrary list
  - A is an arbitrary element of a list

Example 1

- Define (using rex):

```
reverse(L) = reverse2(L, [ ]);

reverse2([ ], M) => M;

reverse2([A | L], M) => reverse2(L, [A | M]);
```

- Show
  $(\forall L)$ length(reverse(L)) = length(L)

- using
  length([ ]) => 0;
  length([A | L]) => 1+length(L);

Example 1

- Here P(L) is
  $length(reverse(L)) = length(L)$

- Structural Induction says it **suffices** to show:
  - P([]): $length(reverse([])) = length([])$
  - $P(L) \rightarrow P([A \mid L])$:

$$length(reverse(L)) = length(L)$$
$$\rightarrow length(reverse([A \mid L])) = length([A \mid L])$$

Example 1

- Structural Induction says it **suffices** to show:
  - P([]): length(reverse([])) = length([])
  - P(L) → P([A | L]):

    length(reverse(L)) = length(L)

    → length(reverse([A | L])) = length([A | L])

- Unfortunately, it will ***not*** be easy to show this directly, because the **inductive** part of the definition is **not** based on **reverse**, it is based on **reverse2**.

Example 1

- So we have to **broaden** the P we are showing to P′:
  - P′(L):
    (∀M) length(reverse2(L, M)) = length(L) + length(M)

- Then specialize to M = [ ]

  reverse(L) = reverse2(L,[]) and
  length([ ]) = 0
  length(L) + 0 = length(L)

  we get the desired P(L):

  length(reverse(L)) = reverse(L).

Broadening

- This broadening requirement is a typical phenomenon in verifying programs.
- It occurs in most induction patterns in some form.
- In some cases it requires creativity.

Example 1: Proof of P′

- Structural Induction says it **suffices** to show:
  - P′([]):
    (∀M) length(reverse2([], M)) = length([]) + length(M)

  - P′(L) → P′([A | L]):
    (∀M) length(reverse2(L, M)) = length(L) + length(M)
    → length(reverse2([A | L], M)) = length([A | L]) + length(M)

- These two parts can be shown separately, as is the case with most inductive proofs.

Example 1: Proof of P′ ′: Basis

- Showing the basis:
  - P′([]):
    (∀M) length(reverse2([], M)) = length([]) + length(M)

- We use **symbolic evaluation**: reverse2([], M) = M
  from the definition of reverse2, and length([]) = 0.

- We have reduced the basis to:
  (∀M) length(M) = 0 + length(M)
  and we can appeal to the definition of + to verify this equality.

Example 1: Proof of P′: Induction Step

- Showing the induction step:
  - $P'(L) \rightarrow P'([A \mid L])$:

$$(\forall M)\ \text{length}(\text{reverse2}(L, M)) = \text{length}(L) + \text{length}(M)$$

$$\rightarrow (\forall N)\ \text{length}(\text{reverse2}([A \mid L], N)) = \text{length}([A \mid L]) + \text{length}(N)$$

- (We use different quantifiers M and N to avoid messing up later.)
- Assume the stmt before the $\rightarrow$, to show the stmt after.

Example 1: Proof of P': Induction Step

- Assume $(\forall M)$ length(reverse2(L, M)) = length(L) + length(M)
  to show:
  $(\forall N)$ length(reverse2([A | L], N)) = length([A | L]) + length(N)

- Again evaluate symbolically:
  reverse2([A | L], N) = reverse2(L, [A | N])

  Now we can view [A | N] as M in the *assumed* equality,
  so length(reverse2(L, [A | N])) = length(L) + length([A | N])
  = length(L) + 1 + length(N) (LHS)

  Also, the rhs of the "to show" is length([A | L]) + length(N)
  = 1 + length(L) +length(N) (RHS)

- Allowing ourselves the commutative law for + then gives the equality to be shown.

Comments

- While we did not use a totally formal system to the derivation, we could have.
- What we did was a little more formal than "…" type of reasoning ("elliptic reasoning").

Structural Induction in Jape

- Load functional_programming theory
- "Proof by clicking" (and **double-clicking** does evaluation/auto-expansion)
- Rules:

  = reflexive (A=A)
  = transitive (A=B AND B=C ⇒ A=C)
  = symmetric (A=B ⇒ B=A)
  A=C AND B=D ⇒ (A,B)=(C,D)
  F x = G x ⇒ F = G
  F (x,y) = G (x, y) ⇒ F = G

  rewrite
  rewrite backwards
  Unfold/Fold with hypothesis
  Unfold with hypothesis
  Fold with hypothesis

  Find
  Flatten

  List Induction — i.e. structural induction
  Boolean cases
  Monoid Definition

Notation

- [] is the empty list
- [a] is the list of one element, a
- ++ is list concatenation or append
- $\bullet$ is function composition:
  $(f \bullet g)x = f(g\ x)$

Induction Model

- The list induction model Jape uses is different from the typical asymmetric one using ‘cons’ i.e. [|] in rex.
- The definition is:
  - [] is a list (the empty list).
  - [a] is a list (the list of one element, a).
  - If x and and y are lists, then so is x++y (the concatenation, or appending, of x followed by y).

Structural Induction in the Jape Model

- To prove a property P for all lists, prove:
  - P([]).
  - P([a]), for any single-element list.
  - If P(x) and P(y), then P(x++y).
- So there are two base cases.
- There are two parts to the induction hypothesis.

Example Proof in Jape

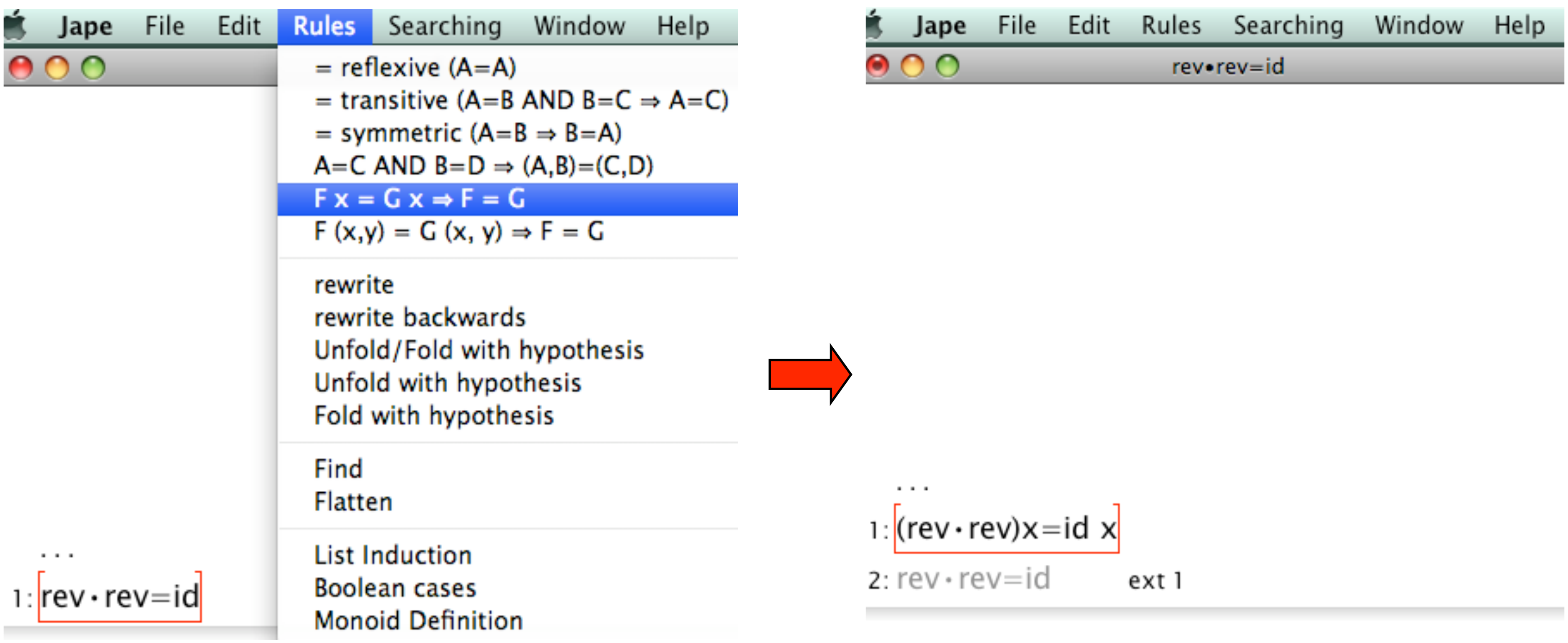
- The reverse of the reverse of a list is the original list itself:
- rev ● rev = id
- recalling that ● is functional composition

Definition of rev

- rev [] = []
- rev [a] = [a]
- rev (x ++ y) = (rev y) ++ (rev x)

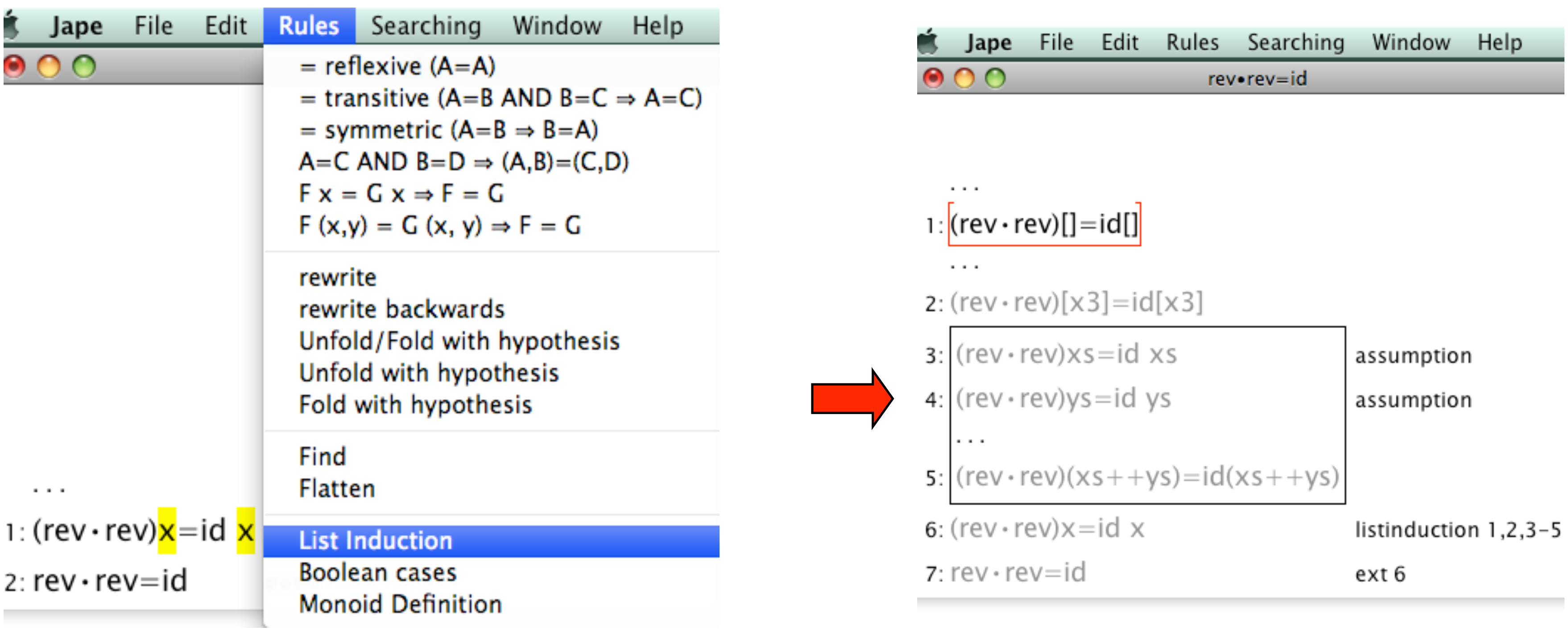
Steps (working backward)

- Introduce a variable representing the argument list:



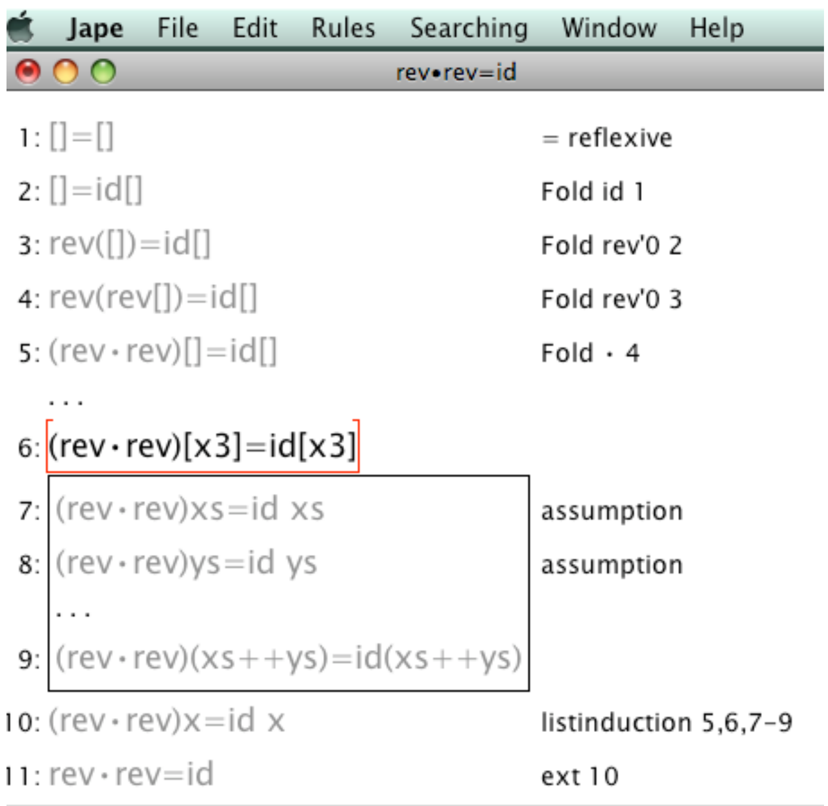
Steps (working backward)

- Text select both arguments and indicate list induction:



Steps (working backward)

- Prove the [] base case by several double-clicks (which search for applicable rules).



[] base case done

"Fold" rule means to replace function app with corresponding definition, working backward.

Steps (working backward)

- Prove the [x] base case by several double-clicks (which search for applicable rules).

rev•rev=id

1:	[]=[]	= reflexive
2:	[]=id[]	Fold id 1
3:	rev([])=id[]	Fold rev'0 2
4:	rev(rev[])=id[]	Fold rev'0 3
5:	(rev · rev)[]=id[]	Fold · 4
6:	[x3]=[x3]	= reflexive
7:	[x3]=id[x3]	Fold id 6
8:	rev([x3])=id[x3]	Fold rev'1 7
9:	rev(rev[x3])=id[x3]	Fold rev'1 8
10:	(rev · rev)[x3]=id[x3]	Fold · 9
11:	(rev · rev)xs=id xs	assumption
12:	(rev · rev)ys=id ys	assumption
	. . .	
13:	(rev · rev)(xs++ys)=id(xs++ys)	
14:	(rev · rev)x=id x	listinduction 5,10,11-13
15:	rev · rev=id	ext 14

[x] base case done

"Fold" rule means to replace function app with corresponding definition, working backward.

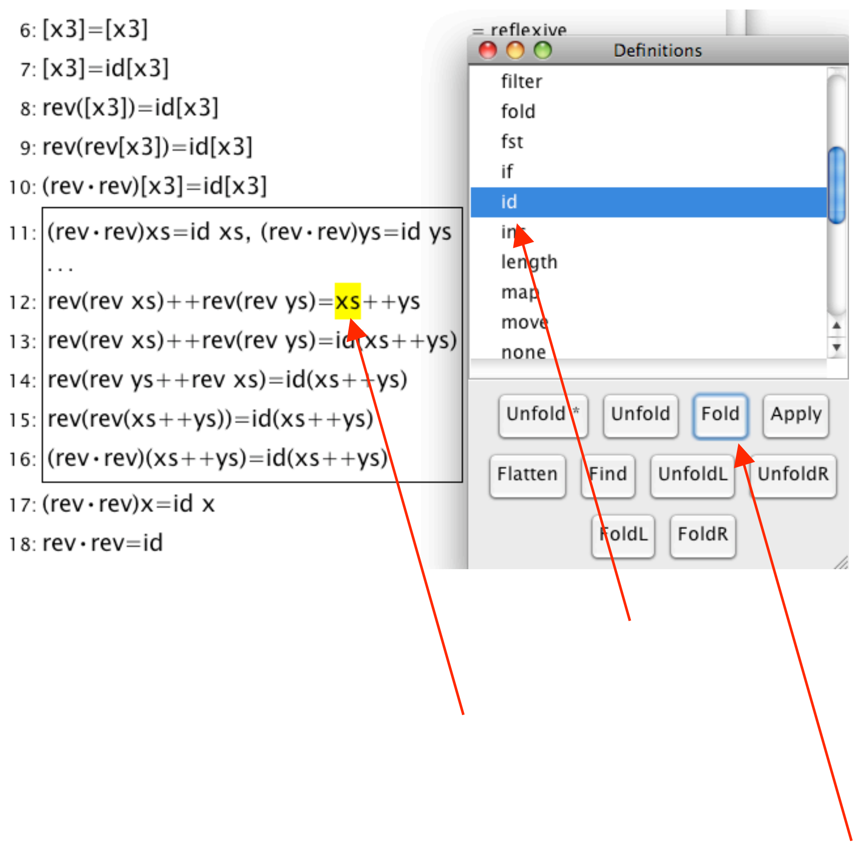
Steps (working backward)

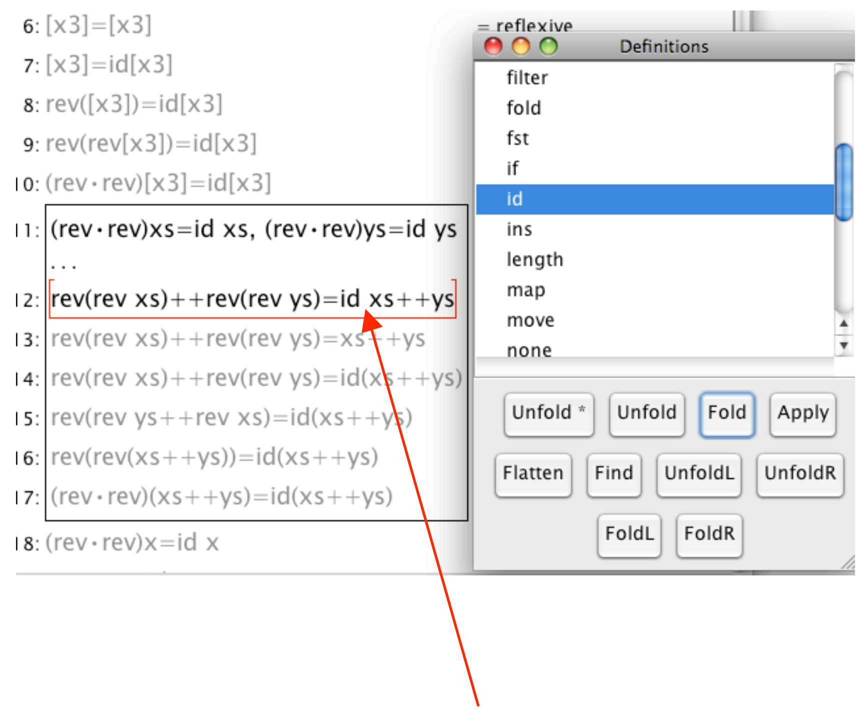
- Prove the induction step. The first few steps can be done automatically.

11:	(rev · rev)xs=id xs, (rev · rev)ys=id ys	assumptions
	. . .	
12:	$\boxed{\text{rev(rev xs)++rev(rev ys)=xs++ys}}$	
13:	rev(rev xs)++rev(rev ys)=id(xs++ys)	Fold id 12
14:	rev(rev ys++rev xs)=id(xs++ys)	Fold rev'2 13
15:	rev(rev(xs++ys))=id(xs++ys)	Fold rev'2 14
16:	(rev · rev)(xs++ys)=id(xs++ys)	Fold · 15
17:	(rev · rev)x=id x	listinduction 5,10,11–16
18:	rev · rev=id	ext 17

Steps (working backward)

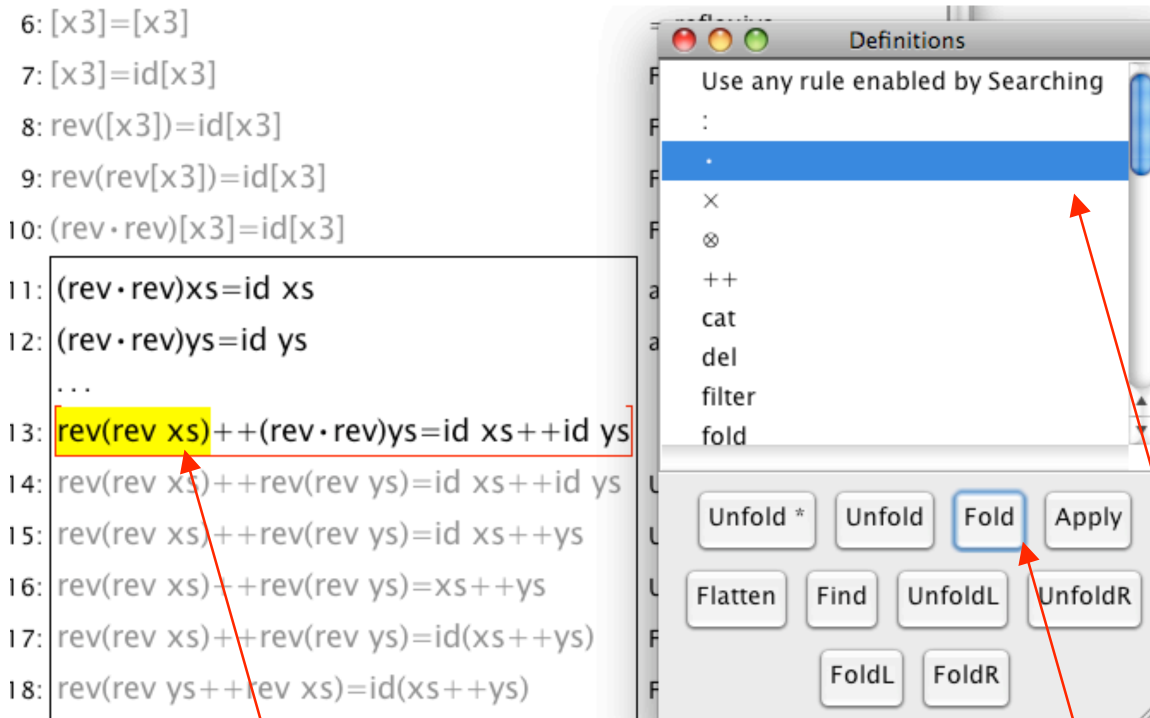
- From here on, we need to be more explicit, apparently:





Steps (working backward)

- More explicit steps



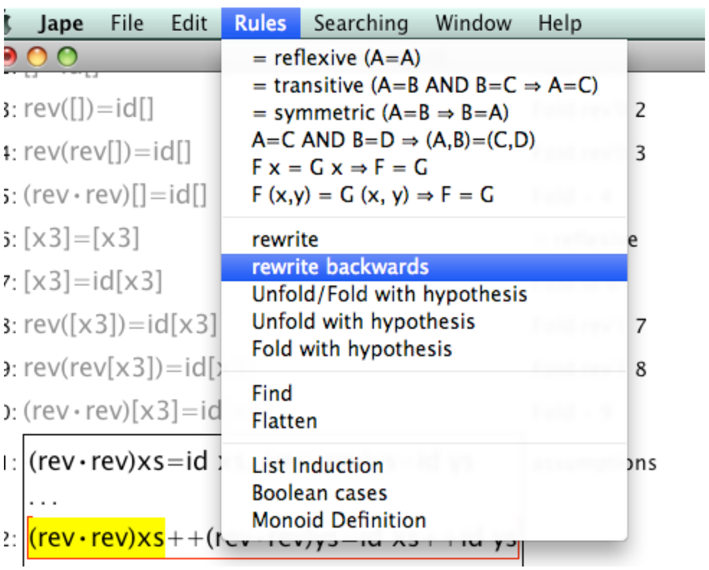
Steps (working backward)

- Obviously we are on the right track. How to close?

11:	(rev · rev)xs=id xs, (rev · rev)ys=id ys	assumptions
	...	
12:	(rev · rev)xs++(rev · rev)ys=id xs++id ys	
13:	rev(rev xs)++(rev · rev)ys=id xs++id ys	Unfold using · 12
14:	rev(rev xs)++rev(rev ys)=id xs++id ys	Unfold using · 13
15:	rev(rev xs)++rev(rev ys)=id xs++ys	Unfold using id 14
16:	rev(rev xs)++rev(rev ys)=xs++ys	Unfold using id 15
17:	rev(rev xs)++rev(rev ys)=id(xs++ys)	Fold id 16
18:	rev(rev ys++rev xs)=id(xs++ys)	Fold rev'2 17
19:	rev(rev(xs++ys))=id(xs++ys)	Fold rev'2 18
20:	(rev · rev)(xs++ys)=id(xs++ys)	Fold · 19

Steps (working backward)

- Is this the best way? Now need to unify explicitly. Repeat two steps.



11:	(rev·rev)xs=id xs, (rev·rev)ys=id ys	assumptions
	...	
12:	_X=(rev·rev)xs	
	...	
13:	_X++(rev·rev)ys=id xs++id ys	
14:	(rev·rev)xs++(rev·rev)ys=id xs++id ys	rewritebackwards 12,13

11:	(rev·rev)xs=id xs, (rev·rev)ys=id ys	assumptions
	...	
12:	id xs=(rev·rev)xs	
	...	
13:	id ys=(rev·rev)ys	
14:	id xs++id ys=id xs++id ys	= reflexive
15:	id xs++(rev·rev)ys=id xs++id ys	rewritebackwards 13,14
16:	(rev·rev)xs++(rev·rev)ys=id xs++id ys	rewritebackwards 12,15

Steps (working backward)

- Now close by double-clicking a couple of times:

11:	(rev · rev)xs=id xs, (rev · rev)ys=id ys	assumptions
12:	id xs=id xs	= reflexive
13:	id xs=(rev · rev)xs	Fold with hypothesis 11.1,12
14:	id ys=id ys	= reflexive
15:	id ys=(rev · rev)ys	Fold with hypothesis 11.2,14
16:	id xs++id ys=id xs++id ys	= reflexive
17:	id xs++(rev · rev)ys=id xs++id ys	rewritebackwards 15,16
18:	(rev · rev)xs++(rev · rev)ys=id xs++id ys	rewritebackwards 13,17
19:	rev(rev xs)++(rev · rev)ys=id xs++id ys	Unfold using · 18
20:	rev(rev xs)++rev(rev ys)=id xs++id ys	Unfold using · 19
21:	rev(rev xs)++rev(rev ys)=id xs++ys	Unfold using id 20
22:	rev(rev xs)++rev(rev ys)=xs++ys	Unfold using id 21
23:	rev(rev xs)++rev(rev ys)=id(xs++ys)	Fold id 22
24:	rev(rev ys++rev xs)=id(xs++ys)	Fold rev'2 23
25:	rev(rev(xs++ys))=id(xs++ys)	Fold rev'2 24
26:	(rev · rev)(xs++ys)=id(xs++ys)	Fold · 25

A Shorter Proof

(obtained by deferring use of induction a couple of steps)

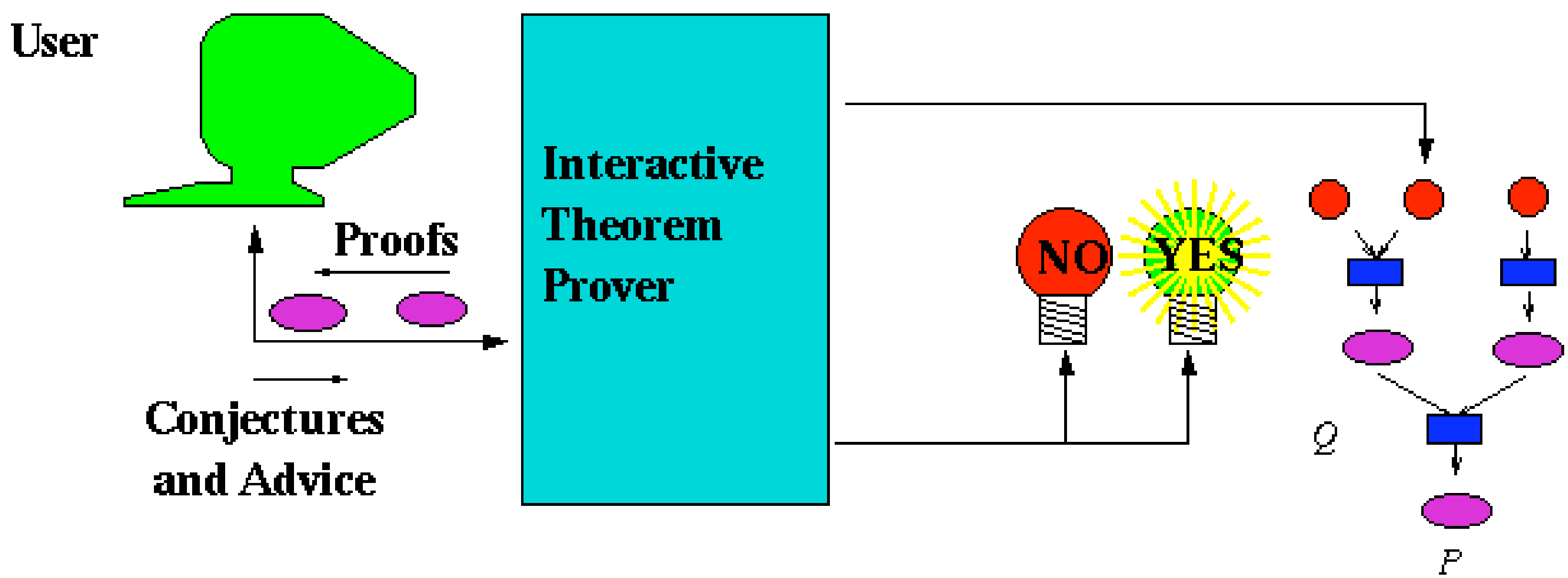
1:	[]=[]	= reflexive
2:	rev([])=[]	Fold rev'0 1
3:	rev(rev[])=[]	Fold rev'0 2
4:	[x5]=[x5]	= reflexive
5:	rev([x5])=[x5]	Fold rev'1 4
6:	rev(rev[x5])=[x5]	Fold rev'1 5
7:	rev(rev xs)=xs, rev(rev ys)=ys	assumptions
8:	(rev(xs++ys))=rev ys++rev xs	rev'2
9:	xs++ys=xs++ys	= reflexive
10:	xs++rev(rev ys)=xs++ys	Fold with hypothesis 7.2,9
11:	rev(rev xs)++rev(rev ys)=xs++ys	Fold with hypothesis 7.1,10
12:	rev(rev ys++rev xs)=xs++ys	Fold rev'2 11
13:	rev(rev(xs++ys))=xs++ys	[rewrite] 8,12
14:	rev(rev x)=x	listinduction 3,6,7-13
15:	rev(rev x)=id x	Fold using id 14
16:	(rev·rev)x=id x	Fold · 15
17:	rev·rev=id	ext 16

Once the placement
of induction is settled
(must text-select both x's)
the rest can be done
by double-clicks.

More Automation
(CL: Computational Logic)

- Automated tools such as ACL2 can be used to do this form of proof on a computer.
- ACL2 = "Applicative Common Lisp 2"
- ACL2 is an interactive theorem prover based on Lisp and structural induction
- Originally called the Boyer/Moore Theorem Prover, this designed has been refined continuously since 1970.
- See:
  http://www.cs.utexas.edu/users/moore/acl2/

Theorem Prover
Use-Case Diagram



Proof

ACL2 includes

- **Ordinary** Lisp execution
- **Symbolic** execution (unfolding, rewriting)
- Automated theorem proving
- Formalism for **admitting axioms** to the system

Sample Function Definition in ACL2

```
ACL2 !>
(defun app (x y)
  (cond ((endp x) y)
        (t (cons (car x)
                 (app (cdr x) y)))))
```

endp checks for the list being empty

The rex equivalent is:

app([ ], Y) => Y;
app([A | X], Y) => [A | app(X, Y)];

Sample ACL Ordinary Evaluations

```
ACL2 !>(app nil '(x y z))
(X Y Z)


ACL2 !>(app '(1 2 3) '(4 5 6 7))
(1 2 3 4 5 6 7)


ACL2 !>(app '(a b c d e f g) '(x y z))
(a b c d e f g x y z)


ACL2 !>(app (app '(1 2) '(3 4)) '(5 6))
(1 2 3 4 5 6)
```

Sample Property to be Proved

A theorem that asserts that function app is associative:

```
ACL2!>
     (defthm associativity-of-app
        (equal (app (app a b) c)
               (app a (app b c))))
```

This can be proved using Structural Induction

- (equal (app (app a b) c)
  (app a (app b c))))

- In other words,
  $(\forall a)(\forall b)(\forall c)\ app(app(a, b), c) = app(a, app(b, c))$

- Exercise: Try proving this by hand.

ACL2 Theorem Prover Narrative Output (1)

```
(defthm associativity-of-app
   (equal (app (app a b) c)
          (app a (app b c))))
```

Name the formula above *1.

Perhaps we can prove *1 by induction. Three induction schemes are suggested by this conjecture. Subsumption reduces that number to two. However, one of these is flawed and so we are left with one viable candidate.

(continued)

ACL2 Theorem Prover Output (2)

We will induct according to a scheme suggested by (APP A B). If we let (:P A B C) denote *1 above then the induction scheme we'll use is

```
(AND
  (IMPLIES (AND (NOT (ENDP A))
                (:P (CDR A) B C))
           (:P A B C))
  (IMPLIES (ENDP A) (:P A B C))).
```

This induction is justified by the same argument used to admit APP, namely, the measure (ACL2-COUNT A) is decreasing according to the relation E0-ORD-< (which is known to be well-founded on the domain recognized by E0-ORDINALP). When applied to the goal at hand the above induction scheme produces the following two nontautological subgoals.

ACL2 Theorem Prover Output (3)
Simplification of the Induction Step

Subgoal *1/2

```
(IMPLIES (AND (NOT (ENDP 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)))).
```

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/2'

```
(IMPLIES (AND (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)))).
```

But simplification reduces this to T, using the :definition APP, the :rewrite rules CDR-CONS and CAR-CONS and primitive type reasoning.

ACL2 Theorem Prover Output (5)
Simplification of the Basis

Subgoal *1/1
```
(IMPLIES (ENDP A)
        (EQUAL (APP (APP A B) C)
              (APP A (APP B C)))).
```

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
```
(IMPLIES (NOT (CONSP A))
        (EQUAL (APP (APP A B) C)
              (APP A (APP B C)))).
```

But simplification reduces this to T, using the :definition APP and primitive type reasoning.

i.e. True

Using Results

- Once the theorem is proved, it is saved in the system to be used as a **rewrite rule**.

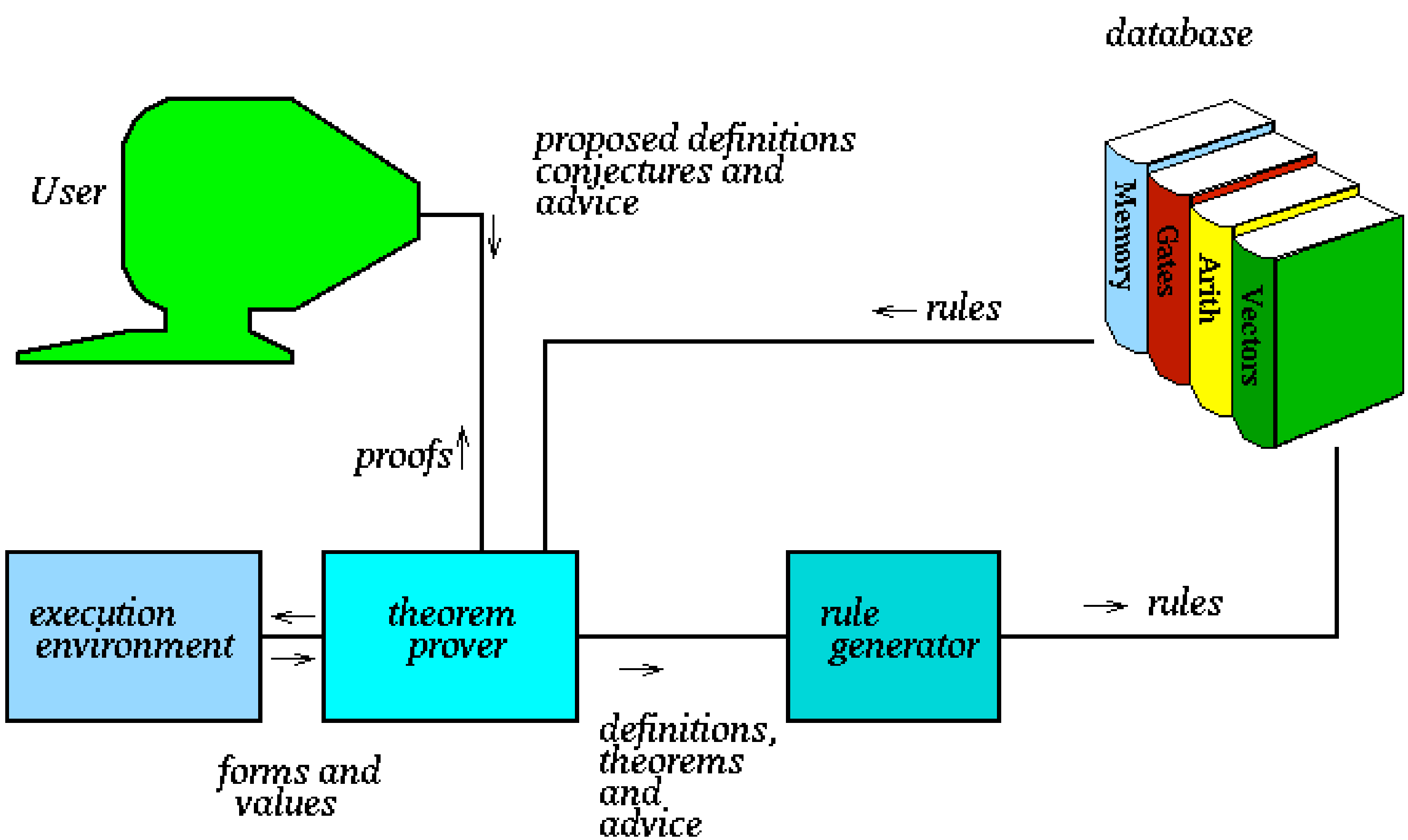
- The system can henceforth rewrite

  (app (app x y) z)

  as

  (app x (app y z))

ACL2 System Architecture



Proved by ACL2

- the **gate-level design** of an academic microprocessor --- which was then **fabricated** and tested,
- **a compiler, an assembler, a linker, and a loader** for the above microprocessor,
- **application programs** written in the source language of the above compiler
- the ``CLI stack'' -- the chaining together of the above theorems to **establish correctness of applications running on a fabricated chip**,
- 21 of the 22 routines in the **Berkeley C String library** (when compiled by `gcc -o` for the Motorola 68020),
- **microcode** programs extracted from the ROM of the Motorola CAP digital signal processor,
- the **microcode** for floating-point division and square root on the AMD K5,
- the **RTL** implementing each of the elementary floating-point operations on the **AMD Athlon**,
- **safety-critical code** involved in trainborne control software written by Union Switch and Signal,
- components of the Rockwell-Collins Avionics JEM1, **the world's first Java virtual machine in silicon**, and
- bootstrapping code for the IBM 4758 **secure co-processor**.