Proving Programs by Structural Induction

Verification Patterns

- **Structural induction** *on program structure* involves proving properties based on the manner in which programs are composed.
- **Transition induction** involves proving properties base on the number of *steps* executed.
- **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 not much generality 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:

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

Structural Induction on Lists

- $$\frac{P(\,[\;]\,),\quad P(L) \rightarrow P(\,[A \mid L]\,)}{(\forall L)\; P(L)}$$

- 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) → P([A | L]):

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

    → length(reverse([A | L])) = length([A | 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 for the special case M = [ ]
  reverse(L) = reverse2(L,[]) and
  length([ ]) = 0
  we get 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 beyond what can reasonably be automated.

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:
  - $P'(L) \rightarrow P'([A \mid L])$:

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

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

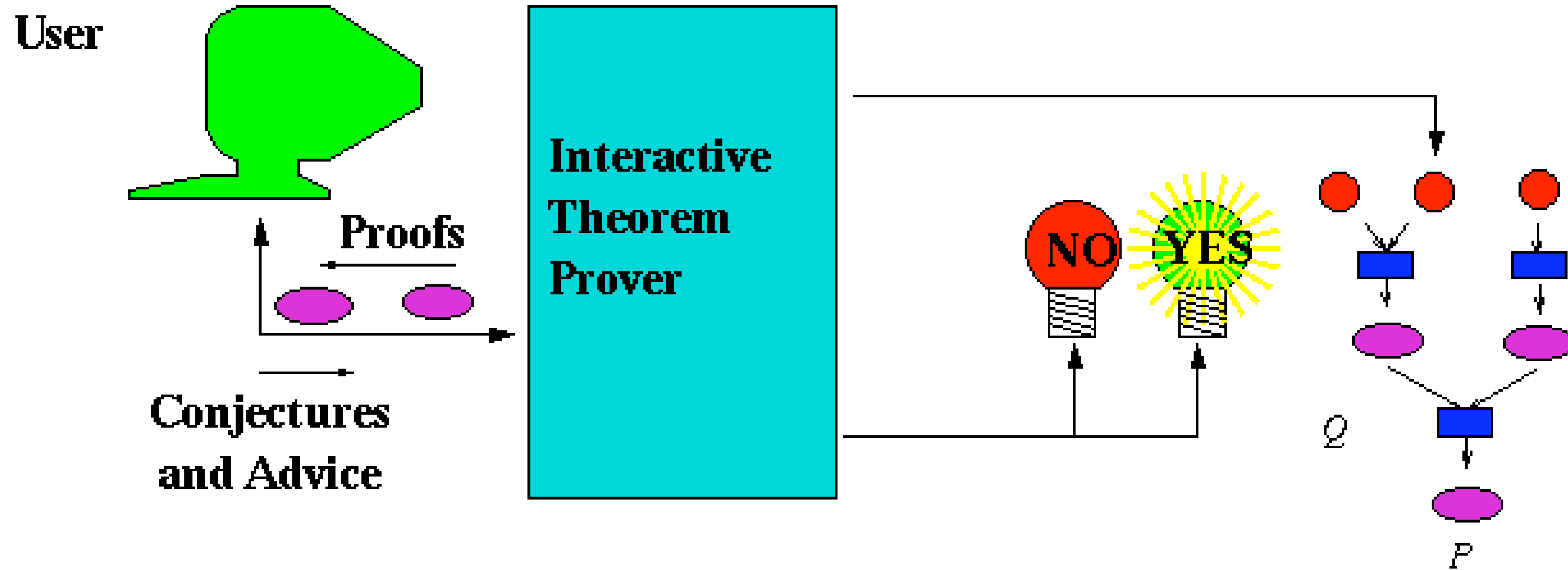
- While we did not use a totally formal system to the derivation, we could have.
- What we did was a little more formal than "…" type of reasoning ("elliptic reasoning").

Automation

- 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



ACL2 includes

- **Ordinary** Lisp execution
- **Symbolic** execution
- 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: Prove 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)

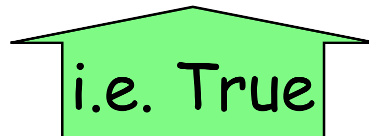
Simplification of the Basis

```
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.



Using Results

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

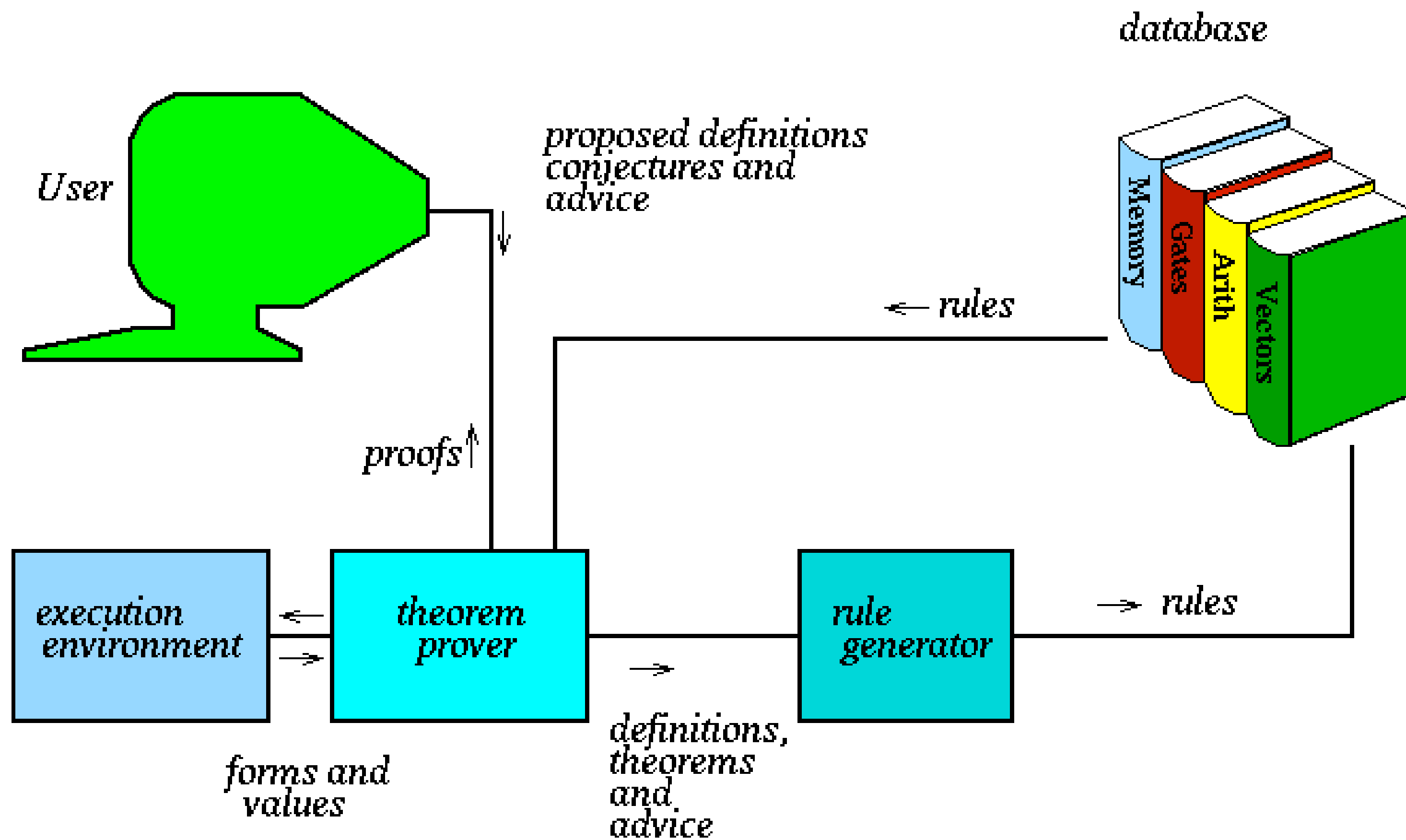
- The system can henceforth rewrite

  (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**.

Induction on Program Structure