Induction in Coq

Prof. Clarkson
Fall 2017

Today's music: *Pictures of Pandas Painting* by They Might Be Giants
Review

Previously in 3110:
• Functional programming in Coq
• Logic in Coq
• Curry-Howard correspondence (proofs are programs)

Today:
• Induction in Coq
REVIEW:
INDUCTION ON NATURAL NUMBERS
Structure of inductive proof

Theorem:
for all natural numbers $n$, $P(n)$.

Proof: by induction on $n$

Case: $n = 0$
Show: $P(0)$

Case: $n = k+1$
IH: $P(k)$
Show: $P(k+1)$

QED
**Sum to n**

```plaintext
let rec sum_to n =
    if n=0 then 0
    else n + sum_to (n-1)
```

**Theorem:**
for all natural numbers n,

\[ \sum_{i=0}^{n} i \]

**Proof:** by induction on n

\[ P(n) \equiv (\text{sum_to n} = n \times (n+1) / 2) \]
Base case

Case: \( n = 0 \)

Show:

\[
P(0) \
\equiv \text{sum}_\text{to} \ 0 = 0 \cdot (0+1) / 2 \\
\equiv 0 = 0 \cdot (0+1) / 2 \\
\equiv 0 = 0
\]

let rec sum_to n =
  if n=0 then 0
  else n + sum_to (n-1)
Inductive case

Case: \( n = k + 1 \)

IH: \( P(k) \equiv \text{sum}_{\text{to}} k = k \times (k+1) / 2 \)

Show:

\[
\begin{align*}
P(k+1) & \equiv \text{sum}_{\text{to}} (k+1) = (k+1) \times (k+2) / 2 \\
& \equiv (k+1) + \text{sum}_{\text{to}} (k+1-1) = (k+1) \times (k+2) / 2 \\
& \equiv (k+1) + \text{sum}_{\text{to}} k = (k+1) \times (k+2) / 2 \\
& \equiv (k+1) + k \times (k+1) / 2 = (k+1) \times (k+2) / 2
\end{align*}
\]

and that holds by algebraic reasoning

QED
Yup, induction

When your instructor wants you to use induction.
INDUCTION ON LISTS
Structure of inductive proof

Theorem: for all natural numbers n, P(n).

Proof: by induction on n

Case: n = 0
Show: P(0)

Case: n = k+1
IH: P(k)
Show: P(k+1)

QED
Structure of inductive proof

Theorem: for all lists lst, P(lst).

Proof: by induction on lst

Case: lst = []
Show: P([])

Case: lst = h::t
IH: P(t)
Show: P(h::t)

QED
Append nil

let rec (@) lst1 lst2 =
  match lst1 with
  | [] -> lst2
  | h::t -> h ::: (t @ lst2)

Theorem:
for all lists lst, lst @ [] = lst.

Proof: by induction on lst

P(lst) \equiv lst @ [] = lst
Base case

Case: \( lst = [] \)

Show:

\[
P([])
\equiv [] @ [] = []
\equiv [] = []
\]
Inductive case

\[ P(lst) \equiv lst @ [] = lst \]

Case: \( lst = h::t \)

IH: \( P(t) \equiv t @ [] = t \)

Show:
\[
\begin{align*}
P(h::t) & \equiv (h::t) @ [] = h::t \\
& \equiv h::(t @ []) = h::t \\
& \equiv h::t = h::t
\end{align*}
\]

QED
Append nil in Coq

Theorem app_nil :
    forall (A:Type) (lst : list A),
    lst ++ nil = lst.
Proof.
    intros A lst.
    induction lst as [ | h t IH].
    - trivial.
    - simpl. rewrite -> IH. trivial.
Qed.
Theorem app_nil :
  forall (A:Type) (lst : list A),
  lst ++ nil = lst.
Proof.
  intros A lst.
  induction lst as [ | h t IH].
  - trivial.
  - simpl. rewrite -> IH. trivial.
Qed.

++ is append operator in Coq
base case: nothing to name
inductive case: name head, tail, and inductive hypothesis
rewrite -> tactic replaces LHS of equality with RHS
Append is associative

Theorem app_assoc :
  forall (A:Type) (l1 l2 l3 : list A),
  l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3.

Proof.
  intros A l1 l2 l3.
  induction l1 as [ | h t IH].
  - trivial.
  - simpl. rewrite -> IH. trivial.
Qed.
INDUCTION ON NATS
Inductive types

Induction works on inductive types, e.g.

Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A -> list A -> list A

Need an inductive definition of natural numbers...
Naturals

Inductive nat : Set :=
  | O : nat       (* zero *)
  | S : nat -> nat (* succ *)

type nat = O | S of nat

0 is O
1 is S 0
2 is S (S 0)
3 is S (S (S 0))

• unary representation
• Peano arithmetic
Induction on nat(ural)s

Theorem:
for all n: nat, P(n)

Proof: by induction on n

Case: n = 0
Show: P(0)

Case: n = S k
IH: P(k)
Show: P(S k)

QED
Goal: redo this proof in Coq

```
let rec sum_to n =
  if n=0 then 0
  else n + sum_to (n-1)
```

Theorem:
for all natural numbers n,
sum_to n = n * (n+1) / 2.

Proof: by induction on n
Defining sum_to

Fixpoint sum_to (n:nat) : nat :=
  if n = 0 then 0
  else n + sum_to (n-1).

Error: The term "n = 0" has type "Prop" which is not a (co-)inductive type.

Fixpoint sum_to (n:nat) : nat :=
  if n =? 0 then 0
  else n + sum_to (n-1).

Recursive definition of sum_to is ill-formed.

Recursive call to sum_to has principal argument equal to "n - 1" instead of a subterm of "n".
No infinite loops

Fixpoint inf (x:nat) : nat :=
  inf x.

Recursive definition of inf is ill-formed.

... Recursive call to inf has principal argument equal to "x" instead of a subterm of "x".
Why no infinite loops?

In OCaml:

# let rec inf x = inf x
val inf : 'a -> 'b = <fun>

By propositions-as-types, these are the same:

• 'a -> 'b
• A ⇒ B

What if A=True, B=False?

Infinite loops prove False!
Defining sum_to

Fixpoint sum_to (n:nat) : nat :=
    match n with
    | 0 => 0
    | S k => n + sum_to k
end.

sum_to is defined

k is a subterm of n, because n = S k,
Sum to n in Coq

Theorem sum_sq_no_div : 
  forall n : nat, 
  2 * sum_to n = n * (n+1).
Proof.
  intros n.
  induction n as [ | k IH].
  - trivial.
  - rewrite -> sum_helper.
    rewrite -> IH.
    ring.
Qed.
Lemma sum_helper :
    forall n : nat,
    2 * sum_to (S n) = 2 * S n + 2 * sum_to n.
Proof.
    intros n. simpl. ring.
Qed.
Induction and recursion

• Intense similarity between inductive proofs and recursive functions on variants
  – In proofs: one case per constructor
  – In functions: one pattern-matching branch per constructor
  – In proofs: uses IH on "smaller" value
  – In functions: uses recursive call on "smaller" value

• Proofs = programs
• Inductive proofs = recursive programs
Upcoming events

• [next Wed] MS1 due

This is pictures of pandas painting.

THIS IS 3110