Review

Previously in 3110:
• Behavioral equivalence
• Proofs of correctness by induction on naturals

Today:
• Induction on lists
• Induction on trees
Review: Induction on natural numbers

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
Induction principle

for all properties \( P \) of natural numbers, 
if \( P(0) \) 
and (for all \( n \), 
    \( P(n) \) implies \( P(n+1) \)) 
then (for all \( n \), \( P(n) \))
Induction principle

for all properties $P$ of lists,
if $P([])$
and (for all $x$ and $xs$,
    $P(xs)$ implies $P(x::xs)$)
then (for all $xs$, $P(xs)$)
Induction on lists

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

let rec length = function
  | [] -> 0
  | _::xs -> 1 + length xs

let rec append xs1 xs2 = match xs1 with
  | [] -> xs2
  | h::t -> h :: append t xs2

Theorem.
for all lists xs and ys,
  length (append xs ys) ~ length xs + length ys.
Append

Theorem.
for all lists \(xs\) and \(ys\),

\[
\text{length } (\text{append } xs \ ys) \sim \text{length } xs + \text{length } ys.
\]

Proof:  by induction on \(xs\)

Case:  \(xs = []\)
Show:  for all \(ys\),

\[
\text{length } (\text{append } [] \ ys) \sim \text{length } [] + \text{length } ys
\]

\[
\text{length } (\text{append } [] \ ys)
\sim \text{length } ys \quad \text{(eval)}
\sim 0 + \text{length } ys \quad \text{(math)}
\sim \text{length } [] + \text{length } ys \quad \text{(eval,symm.)}
\]
Append

Theorem.
for all lists $xs$ and $ys$,

$$\text{length} \ (\text{append} \ xs \ ys) \sim \text{length} \ xs + \text{length} \ ys.$$ 

Proof: by induction on $xs$

Case: $xs = h::t$
Show: for all $ys$, $\text{length} \ (\text{append} \ (h::t) \ ys) 
\sim \text{length} \ (h::t) + \text{length} \ ys$

IH: ??
If we're trying to prove
for all lists \(xs\) and \(ys\),

\[
\text{length (append } xs \; ys) \sim \text{length } xs + \text{length } ys.
\]

by induction on \(xs\), in the case where \(xs = h::t\), what is the inductive hypothesis?

A. for all \(ys\),

\[
\text{length (append } xs \; ys) \sim \text{length } xs + \text{length } ys
\]

B. for all \(ys\),

\[
\text{length (append } t \; ys) \sim \text{length } t + \text{length } ys
\]

C. for all \(ys\),

\[
\text{length (append } (h::t) \; ys) \\
\sim \text{length } (h::t) + \text{length } ys
\]

D. for all \(h'\) and \(t'\),

\[
\text{length (append } (h::t) \; (h'::t')) \\
\sim \text{length } (h::t) + \text{length } (h'::t')
\]

E. for all \(xs\),

\[
\text{length (append } xs \; t) \sim \text{length } xs + \text{length } t
\]
If we're trying to prove
for all lists $xs$ and $ys$,

\[ \text{length } (\text{append } xs \text{ ys}) \sim \text{length } xs + \text{length } ys. \]

by induction on $xs$, in the case where $xs = h::t$, what is the inductive hypothesis?

A. for all $ys$,
   \[ \text{length } (\text{append } xs \text{ ys}) \sim \text{length } xs + \text{length } ys \]

B. for all $ys$,
   \[ \text{length } (\text{append } t \text{ ys}) \sim \text{length } t + \text{length } ys \]

C. for all $ys$,
   \[ \text{length } (\text{append } (h::t) \text{ ys}) \sim \text{length } (h::t) + \text{length } ys \]

D. for all $h'$ and $t'$,
   \[ \text{length } (\text{append } (h::t) (h':::t')) \sim \text{length } (h::t) + \text{length } (h':::t') \]

E. for all $xs$,
   \[ \text{length } (\text{append } xs \text{ t}) \sim \text{length } xs + \text{length } t \]
Append

Theorem.
for all lists xs and ys,
  \text{length} \ (\text{append} \ xs \ ys) \sim \text{length} \ xs + \text{length} \ ys.

Proof: by induction on xs

Case: \ xs = h::t
Show: for all ys, \text{length} \ (\text{append} \ (h::t) \ ys)
      \sim \text{length} \ (h::t) + \text{length} \ ys
IH: for all ys, \text{length} \ (\text{append} \ t \ ys)
      \sim \text{length} \ t + \text{length} \ ys
Append

Case: xs is h::t

Show: for all ys, length (append (h::t) ys) 
      ~ length (h::t) + length ys

IH: for all ys, length (append t ys) 
    ~ length t + length ys

\[
\begin{align*}
\text{length (append (h::t) ys)} \\
\sim \text{length (h:: append t ys)} & \quad \text{(eval)} \\
\sim 1 + \text{length (append t ys)} & \quad \text{(eval)} \\
\sim 1 + \text{length t} + \text{length ys} & \quad \text{(IH, congr.)} \\
\sim \text{length (h::t) + length ys} & \quad \text{(eval, symm., congr.)}
\end{align*}
\]

QED
Higher-order functions

Proofs about higher-order functions sometimes need an additional axiom:

**Extensionality:**

if (for all \( x \), \((f \ x) \sim (g \ x)\))
then \( f \sim g \)
Compose

let (@@) f g x = f (g x)
let map = List.map

Theorem:
for all functions f and g,
   (map f) @@ (map g) ~ map (f @@ g).

Proof:
By extensionality, we need to show that for all xs,
   ((map f) @@ (map g)) xs ~ map (f @@ g) xs.
By eval, ((map f) @@ (map g)) xs ~ map f (map g xs).
So by transitivity, it suffices to show that
   map f (map g xs) ~ map (f @@ g) xs.
Compose

Show:  \( \text{map } f \ (\text{map } g \ \text{xs}) \sim \text{map } (f \ @@ \ g) \ \text{xs}. \)

Proof: by induction on \( \text{xs} \)

Case: \( \text{xs} = [] \)
Show: \( \text{map } f \ (\text{map } g \ []) \sim \text{map } (f \ @@ \ g) \ [] \)

\[
\begin{align*}
\text{map } f \ (\text{map } g \ []) \\
\sim \ [] \quad \text{(eval)} \\
\sim \text{map } (f \ @@ \ g) \ [] \quad \text{(eval)}
\end{align*}
\]
Compose

Show: \( \text{map } f \ (\text{map } g \ \text{xs}) \sim \text{map } (f \ @@ g) \ \text{xs} \).

Proof: by induction on \( \text{xs} \)

Case: \( \text{xs} = h::t \)

Show: \( \text{map } f \ (\text{map } g \ (h::t)) \sim \text{map } (f \ @@ g) \ \text{t} \)

IH: \( \text{map } f \ (\text{map } g \ \text{t}) \sim \text{map } (f \ @@ g) \ \text{t} \)

\[
\begin{align*}
\text{map } f \ (\text{map } g \ (h::t)) \\
\sim \text{map } f \ ((g \ h)::\text{map } g \ \text{t}) \quad \text{(eval map)} \\
\sim (f \ (g \ h))::\text{map } f \ (\text{map } g \ \text{t}) \quad \text{(eval map)} \\
\sim ((f \ @@ g) \ h)::\text{map } f \ (\text{map } g \ \text{t}) \quad \text{(eval @@)} \\
\sim ((f \ @@ g) \ h)::\text{map } (f \ @@ g) \ \text{t} \quad \text{(IH)} \\
\sim \text{map } (f \ @@ g) \ (h::t) \quad \text{(eval map)}
\end{align*}
\]

Helpful to identify what is being evaluated

let \( @@ \ f \ g \ x = f \ (g \ x) \)
let \( \text{map} = \text{List.map} \)
**Compose**

```
let (@@) f g x = f (g x)
let map = List.map
```

**Theorem:**
for all functions f and g,

\[(\text{map } f) @@ \text{map } g \sim \text{map } (f @@ g).\]

**Proof:**
By extensionality, we need to show that for all \(xs\),

\[(\text{map } f) @@ \text{map } g) \, \text{xs} \sim \text{map } (f @@ g) \, \text{xs}.\]

By eval, \((\text{map } f) @@ \text{map } g) \, \text{xs} \sim \text{map } f \, (\text{map } g \, \text{xs}).\)

So by transitivity, it suffices to show that

\[\text{map } f \,(\text{map } g \, \text{xs}) \sim \text{map } (f @@ g) \, \text{xs}. \text{ We have.} \]

**QED.**
Compose

let (@@) f g x = f (g x)
let map = List.map

Theorem:
for all functions f and g,
   (map f) @@ (map g) ~ map (f @@ g).

Comment: this theorem would be the basis for a nice compiler optimization in a pure language. Replace an operation that processes list twice with an operation that processes list only once.
Trees

d type 'a tree =
    | Leaf
    | Branch of 'a * 'a tree * 'a tree

d let rec reflect = function
    | Leaf -> Leaf
    | Branch(x,l,r) -> Branch(x, reflect r, reflect l)
Trees

reflection of

```
1
/  \
2  3
/ \ / \```

is

```
1
/  \
3  2
/ \ / \```

```
7  6  5  4```
Trees

```ocaml
type 'a tree =
    | Leaf
    | Branch of 'a * 'a tree * 'a tree

let rec reflect = function
    | Leaf -> Leaf
    | Branch(x, l, r) -> Branch(x, reflect r, reflect l)
```

Theorem: for all trees t, reflect(reflect t) ~ t.

Proof: by induction on t.
Induction principle

for all properties $P$ of trees,
  if $P(\text{Leaf})$
  and (for all $x$ and $l$ and $r$,
    $P(l)$ and $P(r)$ implies $P(\text{Branch}(x, l, r))$
  )
  then (for all $t$, $P(t)$)
Induction on trees

Theorem: 
for all trees \( t \), \( P(t) \).

Proof: by induction on \( t \)

Case: \( n = \text{Leaf} \)
Show: \( P(\text{Leaf}) \)

Case: \( n = \text{Branch}(x,l,r) \)
IH: \( P(l) \) and \( P(r) \)
Show: \( P(\text{Branch}(x,l,r)) \)

QED
Trees

Theorem: for all trees t, reflect(reflect t) ~ t.

Proof: by induction on t.

Case: t = Leaf
Show: reflect(reflect Leaf) ~ Leaf

    reflect(reflect Leaf)
~ Leaf                   (eval)
Trees

Theorem: for all trees t, reflect(reflect t) ~ t.

Proof: by induction on t.

Case: t = Branch(x,l,r)

Show:
refl ect(refl ect(Branch(x,l,r))) ~ Branch(x,l,r)

IH: ???

let rec reflect = function
  | Leaf -> Leaf
  | Branch(x,l,r) -> Branch(x, reflect r, reflect l)
Question

How many formulas in inductive hypothesis—i.e., how many inductive hypotheses?

A. 1 (for the Branch constructor)
B. 2 (for the two subtrees)
C. 3 (for the two subtrees and the node's label)
How many formulas in inductive hypothesis—i.e., how many inductive hypotheses?

A. 1 (for the Branch constructor)
B. 2 (for the two subtrees)
C. 3 (for the two subtrees and the node's label)
Theorem: for all trees \( t \), \( \text{reflect}(\text{reflect}(t)) \sim t \).

Proof: by induction on \( t \).

Case: \( t = \text{Branch}(x,l,r) \)

Show:
\[
\text{reflect}(\text{reflect}(\text{Branch}(x,l,r))) \sim \text{Branch}(x,l,r)
\]

IH:
1. \( \text{reflect}(\text{reflect}(l)) \sim l \)
2. \( \text{reflect}(\text{reflect}(r)) \sim r \)
Trees

Show:
\[ \text{reflect}(\text{reflect}(\text{Branch}(x,l,r))) \sim \text{Branch}(x,l,r) \]

IH:
1. \[ \text{reflect}(\text{reflect}(l)) \sim l \]
2. \[ \text{reflect}(\text{reflect}(r)) \sim r \]

\[ \text{reflect}(\text{reflect}(\text{Branch}(x,l,r))) \]
\[ \sim \text{reflect}(\text{Branch}(x, \text{reflect}(r), \text{reflect}(l))) \quad \text{(eval)} \]
\[ \sim \text{Branch}(x, \text{reflect}(\text{reflect}(l)), \text{reflect}(\text{reflect}(r))) \quad \text{(eval)} \]
\[ \sim \text{Branch}(x, l, \text{reflect}(\text{reflect}(r))) \quad \text{(IH 1)} \]
\[ \sim \text{Branch}(x, l, r) \quad \text{(IH 2)} \]

QED
Inductive proofs on variants

\textbf{type} \ t = \text{C1 of t1} \mid \ldots \mid \text{Cn of tn}

Theorem: for all \( x : t \), \( P(x) \)
Proof: by induction on \( x \)

\ldots

Case: \( x = \text{Ci y} \)
IH: \( P(v) \) for any components \( v : t \) of \( y \)
Show: \( P(\text{Ci y}) \)

\ldots

QED
General induction principle

type t = ... | Ci of ti | ...

for all properties P of values of type t,
  if
    (for all Ci,
        (for all y:ti,
            (for all components z:t of y, P(z))
            implies P(Ci y)))
  then
    (for all v:t, P(v))
Naturals

(* unary representation *)

\textbf{type} nat = \textbf{Z} | S of nat

\textbf{Theorem:}

\textbf{for all} n:nat, P(n)

\textbf{Proof:}\n
\textbf{by induction on} n

\textbf{Case:}\n
\textbf{n = Z}

\textbf{Show:}\n
P(Z)

\textbf{Case:}\n
\textbf{n = S k}

\textbf{IH:}\n
P(k)

\textbf{Show:}\n
P(S k)

QED

\textbf{Theorem:}

\textbf{for all naturals} n, P(n)

\textbf{Proof:}\n
\textbf{by induction on} n

\textbf{Case:}\n
\textbf{n = 0}

\textbf{Show:}\n
P(0)

\textbf{Case:}\n
\textbf{x = k+1}

\textbf{IH:}\n
P(k)

\textbf{Show:}\n
P(k+1)

QED
Induction

• The kind of induction we've done today is called **structural induction**
  – Induct on the *structure* of a data type
  – Widely used in programming languages theory

• When naturals are coded up as variants, **weak induction** becomes structural induction

• Both structural induction and weak induction (and strong induction) are instances of a very general kind of induction called **well-founded induction**
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

• Inductive proofs truly are a kind of recursive programming (see Curry-Howard isomorphism)
Upcoming events

• [next Wednesday] MS1 due

This is inductive.

THIS IS 3110