Induction and Recursion

Prof. Clarkson
Fall 2015

Today’s music: *Dream within a Dream*
from the soundtrack to *Inception* by Hans Zimmer
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 $n$, $P(n)$)
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 \( \text{lst} \), \( P(\text{lst}) \).

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

Case: \( n = [ ] \)
Show: \( P([ ]) \)

Case: \( n = h::t \)
IH: \( P(t) \)
Show: \( P(h::t) \)

QED
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 $ys$,

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

Proof: by induction on $xs$

Case: $xs = h::t$
Show: for all $ys$, $\text{length (append } (h::t) \text{ } ys) \\
\sim \text{length } (h::t) + \text{length } ys$
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 \(ys\),
\[
\text{length (append (h::t) } ys) \\
\sim \text{length (h::t) + length } ys
\]
D. for all \(h'\) and \(t'\),
\[
\text{length (append (h::t) (h'::t'))} \\
\sim \text{length (h::t) + length (h'::t')}
\]
E. for all \(xs\),
\[
\text{length (append } xs \ t) \sim \text{length } xs + \text{length } t
\]
Question

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

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

IH: for all \(ys\), length (\(append\ t \ ys\))

\[
\sim \text{length} \ t + \text{length} \ ys
\]

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

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

length (append (h::t) ys) 
~ length (h :: append t ys)       (eval)
~ 1 + length (append t ys)       (eval)
~ 1 + length t + length ys       (IH,congr.)
~ length (h::t) + length ys      (eval, symm., congr.)

QED

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

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

From now on, omit many uses of symm., trans., congr.
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**

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

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

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

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

\[((\text{map } f) \circ (\text{map } g)) \; xs \sim \text{map } (f \circ g) \; xs.\]

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

So by transitivity, it suffices to show that

\[\text{map } f \; (\text{map } g \; xs) \sim \text{map } (f \circ g) \; xs.\]
Compose

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

Proof: by induction on \( xs \)

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

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

\[
\begin{align*}
\text{map } f \ (\text{map } g \ (h::t)) & \sim \text{map } f \ ((g \ h)::\text{map } g \ t) \quad \text{(eval map)} \\
& \sim (f \ (g \ h))::\text{map } f \ (\text{map } g \ t) \quad \text{(eval map)} \\
& \sim ((f \ @@ g) \ h)::\text{map } f \ (\text{map } g \ t) \quad \text{(eval @@)} \\
& \sim ((f \ @@ g) \ h)::\text{map} \ (f \ @@ g) \ t \quad \text{(IH)} \\
& \sim \text{map } (f \ @@ g) \ (h::t) \quad \text{(eval map)}
\end{align*}
\]
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 \(xs\),
\[((\text{map } f) @@ (\text{map } g)) \ xs \sim \text{map } (f @@ g) \ xs.\]
By eval, \[((\text{map } f) @@ (\text{map } g)) \ xs \sim \text{map } f (\text{map } g \ xs)\].
So by transitivity, it suffices to show that
\[\text{map } f (\text{map } g \ xs) \sim \text{map } (f @@ g) \ xs.\]
We have.
QED.
**Compose**

```haskell
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

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)
Trees

reflection of

```
     1
    / \  
   2 3  
  / \ / \  
 4 5 6 7
```

is

```
     1
    / \  
   3 2  
  / \ / \  
 7 6 5 4
```
Trees

define type 'a tree as
  | Leaf
  | Branch of 'a * 'a tree * 'a tree

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

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 = Leaf
Show:  P(Leaf)

Case:  n = Branch(x,l,r)
IH:    P(l) and P(r)
Show:  P(Branch(x,l,r))

QED
Trees

Theorem: for all trees \( t \), \( \text{reflect(\text{reflect } t)} \sim t \).

Proof: by induction on \( t \).

Case: \( t = \text{Leaf} \)
Show: \( \text{reflect(\text{reflect } \text{Leaf})} \sim \text{Leaf} \)

\[
\text{reflect(\text{reflect } \text{Leaf})} \sim \text{Leaf} \quad \text{(eval)}
\]
Trees

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

Proof: by induction on t.

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

Show:
   reflect(reflect(Branch(x, l, r))) ~ Branch(x, l, r)

IH: ???
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)
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)
Trees

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:
 reflect(reflect(Branch(x,l,r))) ~ Branch(x,l,r)

IH:
  1. reflect(reflect l) ~ l
  2. reflect(reflect r) ~ r

  reflect(reflect(Branch(x,l,r)))
~ reflect(Branch(x, reflect r, reflect l)) (eval)
~ Branch(x, reflect(reflect l), reflect(reflect r)) (eval)
~ Branch(x, l, reflect(reflect r)) (IH 1)
~ Branch(x, l, r) (IH 2)

QED
Inductive proofs on variants

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

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

\textbf{QED}
General induction principle

for all properties $P$ of $t$, if

\[(\text{for all } C_i, (\text{for all } y, (\text{for all components } z:t \text{ of } y, P(z)) \implies P(C_i y)))\]

then

\[(\text{for all } t, P(t))\]
Naturals

(* unary representation *)

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

Theorem:
for all \( n : \text{nat} \), \( P(n) \)
Proof: by induction on \( n \)

Case: \( n = Z \)
Show: \( P(Z) \)

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

QED

Theorem:
for all naturals \( n \), \( P(n) \)
Proof: by induction on \( n \)

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

Case: \( x = k+1 \)
IH: \( P(k) \)
Show: \( P(k+1) \)

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**
  – see CS 4110
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, CS 4110)
Upcoming events

• [next Thursday] A5 due, including Async and design phase of project

This is inductive.

THIS IS 3110