CS 3110

Induction and Recursion

Prof. Clarkson
Fall 2016

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:

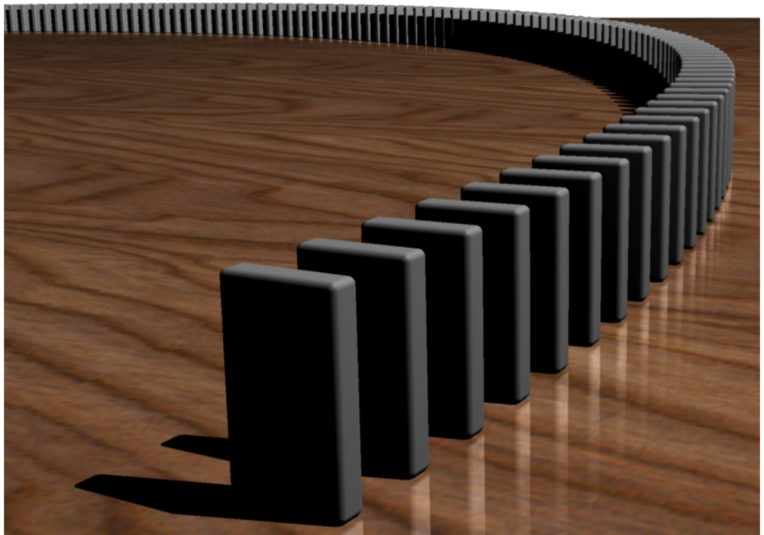
- Induction on lists
- Induction on trees

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
```

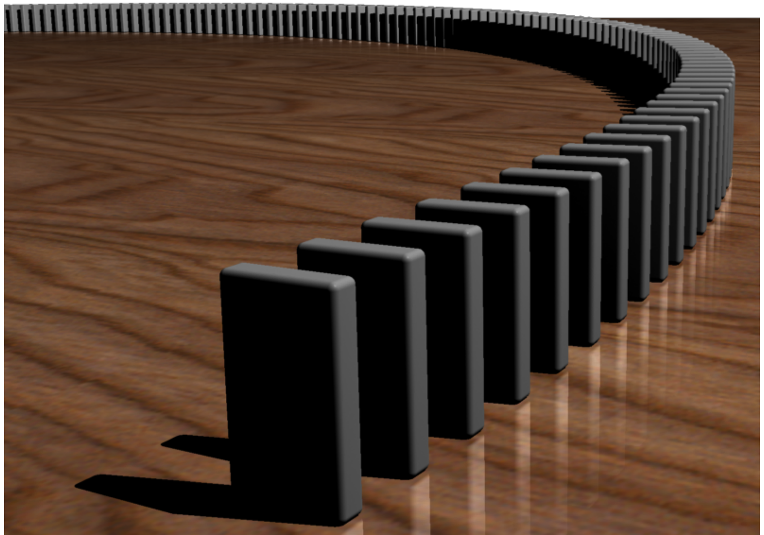
Induction principle

```
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))
```



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

```
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.

Proof:  by induction on xs

Case:  xs = []
Show:  for all ys,
         length (append [] ys) ~ length [] + length ys

  length (append [] ys)
~ length ys                 (eval)
~ 0 + length ys             (math)
~ length [] + length ys     (eval,symm.)
```

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.

Proof:  by induction on xs

Case:  xs = h::t
Show:  for all ys, length (append (h::t) ys)
                    ~ length (h::t) + length ys
IH:    ??
```

Question

If we're trying to prove

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

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

```
A. for all ys,
     length (append xs ys) ~ length xs + length ys
B. for all ys,
     length (append t ys) ~ length t + length ys
C. for all ys,
     length (append (h::t) ys)
     ~ length (h::t) + length ys
D. for all h' and t',
     length (append (h::t) (h'::t'))
     ~ length (h::t) + length (h'::t')
E. for all xs,
     length (append xs t) ~ length xs + length t
```

Question

If we're trying to prove

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

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

```
A. for all ys,
     length (append xs ys) ~ length xs + length ys
B. for all ys,
     length (append t ys) ~ length t + length ys
C. for all ys,
     length (append (h::t) ys)
     ~ length (h::t) + length ys
D. for all h' and t',
     length (append (h::t) (h'::t'))
     ~ length (h::t) + length (h'::t')
E. for all xs,
     length (append xs t) ~ length xs + length t
```

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.

Proof:  by induction on xs

Case:  xs = 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
```

```
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, ...)

QED
```

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) ~ (g x)`)

then `f ~ 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.
```

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

Compose

```
Show:  map f (map g xs) ~ map (f @@ g) xs.

Proof: by induction on xs

Case: xs = []
Show: map f (map g []) ~ map (f @@ g) []

  map f (map g [])
~ []                         (eval)
~ map (f @@ g) []            (eval)
```

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

Compose

```
Show:  map f (map g xs) ~ map (f @@ g) xs.

Proof: by induction on xs

Case: xs = h::t
Show: map f (map g (h::t)) ~ map (f @@ g)
IH:   map f (map g t) ~ map (f @@ g) t

 map f (map g (h::t))
~ map f ((g h)::map g t)          (eval map)
~ (f (g h))::map f (map g t)      (eval map)
~ ((f @@ g) h)::map f (map g t)   (eval @@)
~ ((f @@ g) h)::map (f @@ g) t    (IH)
~ map (f @@ g) (h::t)             (eval map)
```

Helpful to identify what is being evaluated

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.  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

```
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

```
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(Leaf)
  and (for all x and l and r,
         P(l) and P(r) implies P(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
```

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

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)
```

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

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)

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

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

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

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

```
type t = C1 of t1 | ... | Cn of tn

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

...

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

...

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 *)
type nat = Z | S of nat
```

```
Theorem:
  for all n: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**

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