More Proofs By Induction (Trees and General Datatypes)

COS 326
David Walker
Princeton University

notes: http://www.cs.princeton.edu/courses/archive/fall15/cos326/notes/reasoning-data.php

slides copyright 2013-2015 David Walker and Andrew W. Appel
permission granted to reuse these slides for non-commercial educational purposes

Final Exam Schedule Announced

COS 326: ~~July~~ Jan 26, 7:30pm

McCosh 46

Be there or be square.

Exam Topics

In general:

- Anything from the lectures, lecture notes, precepts, assignments
- Precept exercises are a good way to study
- As are past midterms (see Piazza)

Functional programming

- lists, data types, higher-order and polymorphic functions
- map, fold, programming with combinators
- good properties of OCaml

Substitution model, space model, cps, closures, interpreters

Equivalence of programs, proofs by induction

PROOFS ABOUT DATATYPES

Template for Inductive Proofs on Lists

Theorem: For all lists xs, property(xs).

Proof: By induction on lists xs.

Case: xs == [ ]:

... no uses of IH ...

Case: xs == hd :: tl:

IH: property(tl)

Template for Inductive Proofs on Lists

Theorem: For all lists xs, property(xs).

Proof: By induction on lists xs.

Case: xs == [ ]:

... no uses of IH ...

Case: xs == hd :: tl:

IH: property(tl)

one case for empty list

one case for non-empty lists

IH may be used on smaller lists

In general, cases must cover all the lists:

- other possibilities: case for [], case for x1::[], case for x1::x2::tl

Template for Inductive Proofs on Lists

Theorem: For all lists xs, property(xs).

Proof: By induction on lists xs.

Case: xs == [ ]:

... no uses of IH ...

Case: xs == hd :: tl:

IH: property(tl)

one case for empty list

one case for non-empty lists

IH may be used on smaller lists

In general, cases must cover all the lists:

- other possibilities: case for [], case for x1::[], case for x1::x2::tl

just splitting the case for non-empty lists in 2 again

More General Template for Inductive Datatypes

```
type t =  C1 of t1 | C2 of t2 | ... | Cn of tn
```

types t1, t2 ... tn, may contain 1 or more occurrence of t within them.

Examples:

```
type mylist =
  MyNil
| MyCons of int * mylist
```

```
type ‘a tree =
  Leaf
| Node of ‘a * ‘a tree * ‘a tree
```

recursive occurrences

More General Template for Inductive Datatypes

```
type t =  C1 of t1 | C2 of t2 | ... | Cn of tn
```

Theorem: For all x : t, property(x).

Proof: By induction on structure of values x with type t.

More General Template for Inductive Datatypes

type t = C1 of t1 | C2 of t2 | ... | Cn of tn

Theorem: For all x : t, property(x).

Proof: By induction on structure of values x with type t.

Case: x == C1 v:

... use IH on components of v that have type t ...

Case: x == C2 v:

... use IH on components of v that have type t ...

Case: x == Cn v:

... use IH on components of v that have type t ...

A PROOF ABOUT TREES

Another example

```
type ‘a tree = Leaf | Node of ‘a * ‘a tree * ‘a tree

let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

```
type ‘a tree = Leaf | Node of ‘a * ‘a tree * ‘a tree

let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Theorem:
For all (total) functions f : b -> c,
For all (total) functions g : a -> b,
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

"Forall intro"

Theorem:

For all (total) functions f : b -> c,
For all (total) functions g : a -> b,
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

To begin, let's pick an arbitrary total function f and total function g. We'll prove the theorem without assuming any particular properties of f or g (other than the fact that the types match up). So, for the f and g we picked, we'll prove:

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Case: t = Leaf

No inductive hypothesis to use.
(Leaf doesn't contain any smaller components with type tree.)

Proof:

```
   tm f (tm g Leaf)
== tm f Leaf            (eval)
== Leaf                 (eval)
== tm (f <> g) Leaf     (reverse eval)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

Proof:

tm f (tm g (Node (v, l, r)))

== tm (f <> g) (Node (v, l, r))

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

Proof:

tm f (tm g (Node (v, l, r)))
== tm f (Node (g v, tm g l, tm g r)) (eval inner tm)

== tm (f <> g) (Node (v, l, r))

Another example

Theorem:

For all trees t : a tree,

tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l

IH2: tm f (tm g r) == tm (f <> g) r

Proof:

tm f (tm g (Node (v, l, r)))

== tm f (Node (g v, tm g l, tm g r)) (eval inner tm)

Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)

== tm (f <> g) (Node (v, l, r)) (eval reverse)

Another example

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

Proof:

tm f (tm g (Node (v, l, r)))	
== tm f (Node (g v, tm g l, tm g r))	(eval inner tm)
== Node (f (g v), tm f (tm g l), tm f (tm g r))	(eval – since g, tm are total)
Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)	
== tm (f <> g) (Node (v, l, r))	(eval reverse)

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

Proof:

```
   tm f (tm g (Node (v, l, r)))
== tm f (Node (g v, tm g l, tm g r))                (eval inner tm)
== Node (f (g v), tm f (tm g l), tm f (tm g r))     (eval – since g, tm are total)

   Node ((f <> g) v, tm (f <> g) l, tm f (tm g r))
== Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)  (IH2)
== tm (f <> g) (Node (v, l, r))                     (eval reverse)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

Proof:

```
   tm f (tm g (Node (v, l, r)))
== tm f (Node (g v, tm g l, tm g r))                      (eval inner tm)
== Node (f (g v), tm f (tm g l), tm f (tm g r))           (eval – since g, tm are total)
== Node ((f <> g) v, tm f (tm g l), tm f (tm g r))
== Node ((f <> g) v, tm (f <> g) l, tm f (tm g r))        (IH1)
== Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)        (IH2)
== tm (f <> g) (Node (v, l, r))                           (eval reverse)
```

Another example

Theorem:
For all trees t : a tree,
tm f (tm g t) == tm (f <> g) t

Case: t = Node(v, l, r)

IH1: tm f (tm g l) == tm (f <> g) l
IH2: tm f (tm g r) == tm (f <> g) r

```
let rec tm f t =
  match t with
    | Leaf -> Leaf
    | Node (x, l, r) -> Node (f x, tm f l, tm f r)

let (<>) f g =
  fun x -> f (g x)
```

Proof:

tm f (tm g (Node (v, l, r)))	
== tm f (Node (g v, tm g l, tm g r))	(eval inner tm)
== Node (f (g v), tm f (tm g l), tm f (tm g r))	(eval – since g, tm are total)
== Node ((f <> g) v, tm f (tm g l), tm f (tm g r))	(eval reverse)
== Node ((f <> g) v, tm (f <> g) l, tm f (tm g r))	(IH1)
== Node ((f <> g) v, tm (f <> g) l, tm (f <> g) r)	(IH2)
== tm (f <> g) (Node (v, l, r))	(eval reverse)

Summary: Proof Template for Trees

```
type ‘a tree = Leaf | Node of ‘a * ‘a tree * ‘a tree
```

Theorem: For all x : ‘a tree, property(x).

Proof: By induction on the structure of trees x.

Case: x == Leaf:

... no use of inductive hypothesis (this is the smallest tree) ...

Case: x == Node (v, left, right):

IH1: property(left)
IH2: property(right)

... use IH1 and IH 2 in your proof ...

A PROOF ABOUT EXPRESSIONS

A simple expression language

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id
```

A simple expression language

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

let e1 = Add (Int 3, Var "x")
```

A simple expression language

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int
```

A simple expression language

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
   Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
   Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
    Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Int i

eval (opt (Int i))

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Int i

```
   eval (opt (Int i))   (RHS)
== eval (Int i)         (eval of opt)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Int i

```
   eval (opt (Int i))   (RHS)
== eval (Int i)         (eval of opt)
```

case done!
(we reached the LHS from RHS)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

IH: eval (opt e2) == eval e2

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)   IH: eval (opt e2) == eval e2

eval (opt (Add(Int 0, e2))) (LHS)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)   **IH:** eval (opt e2) == eval e2

```
     eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                   (by eval opt)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2) **IH:** eval (opt e2) == eval e2

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
   Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
   Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
    Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

eval (Add(Int 0, e2)) (RHS)

eval (opt (Add(Int 0, e2))) (LHS)
== eval (opt e2) (by eval opt)
== eval e2 (by IH)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

```
    eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                 (by eval opt)
== eval e2                       (by IH)
```

```
   eval (Add(Int 0, e2))          (RHS)
== (eval(Int 0)) + (eval e2)  (eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

```
    eval (opt (Add(Int 0, e2)))  (LHS)
 == eval (opt e2)                (by eval opt)
 == eval e2                      (by IH)
```

```
    eval (Add(Int 0, e2))           (RHS)
 == (eval(Int 0)) + (eval e2)      (eval)
 == 0 + eval e2                    (eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

```
   eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                (by eval opt)
== eval e2                      (by IH)
```

```
   eval (Add(Int 0, e2))            (RHS)
== (eval(Int 0)) + (eval e2)        (eval)
== 0 + eval e2                      (eval)
== eval e2                          (math)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

```
    eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                 (by eval opt)
== eval e2                       (by IH)
```

```
   eval (Add(Int 0, e2))         (RHS)
== (eval(Int 0)) + (eval e2)     (eval)
== 0 + eval e2                   (eval)
== eval e2                       (math)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (ev...
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem: ...xp, eval (opt e) == eval e

case done!
(we showed the
LHS == RHS)

Proof: By induction on the structure ...essions e : exp.

Case: e = Add(Int 0, e2)

```
    eval (opt (Add(Int 0, e2)))  (LHS)
== eval (opt e2)                  (by eval opt)
== eval e2                        (by IH)
```

```
    eval (Add(Int 0, e2))          (RHS)
== (eval(Int 0)) + (eval e2)      (eval)
== 0 + eval e2                     (eval)
== eval e2                         (math)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
   Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
   Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
    Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e2, Int 0)

IH: eval (opt e2) == eval e2

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e2, Int 0)

IH: eval (opt e2) == eval e2

Very similar to the last case – go through it yourself for practice.

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
    Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

IH1: eval (opt e1) == eval e1
IH2: eval (opt e2) == eval e2

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

eval (opt (Add(e1, e2)))   (LHS)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

```
   eval (opt (Add(e1, e2)))       (LHS)
== eval (Add (opt e1, opt e2))   (by eval opt)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

```
   eval (opt (Add(e1, e2)))        (LHS)
== eval (Add (opt e1, opt e2))    (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

eval (Add(e1, e2)) (RHS)

```
   eval (opt (Add(e1, e2)))      (LHS)
== eval (Add (opt e1, opt e2))  (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

```
   eval (opt (Add(e1, e2)))       (LHS)
== eval (Add (opt e1, opt e2))   (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval)
```

```
   eval (Add(e1, e2))          (RHS)
== (eval e1) + (eval e2)  (eval)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Add(e1, e2)

```
   eval (opt (Add(e1, e2)))       (LHS)
== eval (Add (opt e1, opt e2))    (by eval opt)
== eval (opt e1) + eval (opt e2)  (by eval eval)
```

```
   eval (Add(e1, e2))           (RHS)
== (eval e1) + (eval e2)  (eval)
== eval (opt e1) + eval (opt e2)
                (by IH1 and IH2)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eva
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

(opt e) == eval e

case done!
(we showed the
LHS == RHS)

Proof: By induction on the structure of expre ns e : exp.

Case: e = Add(e1, e2)

```
    eval (opt (Add(e1, e2)))        (LHS)
== eval (Add (opt e1, opt e2))     (by eval opt)
== eval (opt e1) + eval (opt e2)   (by eval eval)
```

```
   eval (Add(e1, e2))              (RHS)
== (eval e1) + (eval e2)           (eval)
== eval (opt e1) + eval (opt e2)
                                    (by IH1 and IH2)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Var x

No IH to use because there are no sub-structures with type exp!

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
   Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of expressions e : exp.

Case: e = Var x

```
   eval (opt (Var x))      (LHS)
== eval (Var x)             (by eval opt)
```

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of id

type env
val lookup : env -> id -> int

let rec eval (env: env) (e: exp) : int =
   Int i -> i
| Add (e1, e2) -> (eval env e1) + (eval env e2)
| Var x -> lookup env x
```

```
let rec opt (e:exp) : exp =
   Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| Add (e1,e2) ->
    Add(opt e1, opt e2)
| Var x -> Var x
```

Theorem:
For all e : exp, eval (opt e) == eval e

Proof: By induction on the structure of e

Case: e = Var x

```
   eval (opt (Var x))       (LHS)
== eval (Var x)             (by eval opt)
```

case done!
(we showed the
LHS == RHS)

A simple optimizer

```
type id = string
type exp = Int of int | Add of exp * exp | Var of i

type env
val lookup : e          -> int

let rec eval (env: e
  Int i -> i
| Add (e1
| Var x -> loo
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
| A    ,e2) ->
       (opt e1, opt e2)
       x -> Var x
```

al (opt e) == eval e

PROOF DONE!!!

Proof: P

e!
the
LHS

Case: e = Var x

     eval (opt (Var x))   (LHS
== eval (Var x)   (by   opt)

Summary of Template for Inductive Datatypes

type t = C1 of t1 | C2 of t2 | ... | Cn of tn

Theorem: For all x : t, property(x).

Proof: By induction on structure of values x with type t.

use patterns that divide up the cases

Take inspiration from the structure of the program

Case: x == C1 v:

... use IH on components of v that have type t ...

Case: x == C2 v:

... use IH on components of v that have type t ...

Case: x == Cn v:

... use IH on components of v that have type t ...

Exercise

type ‘a tree = Leaf of ‘a | Node of ‘a tree * ‘a tree

```
let rec flip (t: ‘a tree) =
 match t with
 | Leaf _ -> t
 | Node (a,b) -> Node (flip b, flip a)
```

Theorem: flip(flip t) = t.

Exercise

type ‘a tree = Leaf of ‘a| Node of ‘a tree * ‘a tree

```
let rec flip (t: ‘a tree) =
 match t with
 | Leaf _ -> t
 | Node (a,b) -> Node (flip b, flip a)
```

Theorem: flip(flip t) = t.

Theorem: flip(flip (flip t)) = flip t.

End!