More Proofs By Induction (Trees and General Datatypes)

COS 326

Andrew W. Appel

Princeton University

notes: http://www.cs.princeton.edu/courses/archive/fall23/cos326/notes/reasoning-data.php

Equational Reasoning: Some Key Ideas

What is the fundamental *definition of expression equality* (e1 == e2)?

- two expressions are equal if:
  - they evaluate to equal values, or
  - they both raise the same exception
  - they both fail to terminate
- note: we won't consider expressions that print or have other sorts of I/O or that use mutable data structures

What are some consequences of this definition?

- expression equality is reflexive, symmetric and transitive
- if e1 --> e2 then e1 == e2
- if e1 == e2 then e[e1/x] == e[e2/x]. (substitution of equals for equals)

How do we prove things about recursive functions?

- we use proofs by induction
- to reason about recursive calls on *smaller* data, we assume the property we are trying to prove (ie, we use the *induction hypothesis*)

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = [ ]:

```
   add_all (add_all [] a) b          (LHS of theorem)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = [ ]:

```
   add_all (add_all [] a) b          (LHS of theorem)
== add_all [ ] b                     (by evaluation of add_all)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = [ ]:

```
   add_all (add_all [] a) b          (LHS of theorem)
== add_all [ ] b                     (by evaluation of add_all)
== [ ]                               (by evaluation of add_all)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = [ ]:

add_all (add_all [] a) b	(LHS of theorem)
== add_all [ ] b	(by evaluation of add_all)
== [ ]	(by evaluation of add_all)
== add_all [ ] (a + b)	(by evaluation of add_all)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

```
    add_all (add_all (hd :: tl) a) b          (LHS of theorem)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

```
   add_all (add_all (hd :: tl) a) b          (LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b        (by eval inner add_all)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

```
   add_all (add_all (hd :: tl) a) b            (LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b          (by eval inner add_all)
== (hd+a+b) :: (add_all (add_all tl a) b)      (by eval outer add_all)
==
```

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

add_all (add_all (hd :: tl) a) b	(LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b	(by eval inner add_all)
== (hd+a+b) :: (add_all (add_all tl a) b)	(by eval outer add_all)
== (hd+a+b) :: add_all tl (a+b)	(by IH)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

add_all (add_all (hd :: tl) a) b	(LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b	(by eval inner add_all)
== (hd+a+b) :: (add_all (add_all tl a) b)	(by eval outer add_all)
== (hd+a+b) :: add_all tl (a+b)	(by IH)
== (hd+(a+b)) :: add_all tl (a+b)	(associativity of +)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

Another List example

Theorem: For all lists xs,

add_all (add_all xs a) b == add_all xs (a+b)

Proof: By induction on xs.

case xs = hd :: tl:

add_all (add_all (hd :: tl) a) b	(LHS of theorem)
== add_all ((hd+a) :: add_all tl a) b	(by eval inner add_all)
== (hd+a+b) :: (add_all (add_all tl a) b)	(by eval outer add_all)
== (hd+a+b) :: add_all tl (a+b)	(by IH)
== (hd+(a+b)) :: add_all tl (a+b)	(associativity of +)
== add_all (hd::tl) (a+b)	(by (reverse) eval of add_all)

```
let rec add_all xs c =
  match xs with
  | [ ] -> [ ]
  | hd::tl -> (hd+c)::add_all tl c
```

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 nonempty 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 nonempty 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 tail of nonempty lists in two more cases of the same kind again

PROOFS ABOUT DATATYPES

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 occurrences 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)

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

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))

```
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 (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)
```

```
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

```
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

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 <> 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

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))	
== 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

```
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 (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 ...

PROOFS ABOUT
PROGRAMMING LANGUAGES

You might wonder

We've done some proofs about *individual programs*. eg:

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

```
for all n:int,
   even (2 * n) == true
```

But can we do proofs about entire *programming languages*?

In other words, proofs about *all programs that anyone could ever write in the programming language*?

You might wonder

We've done some proofs about *individual programs*. eg:

```
let rec even n =
  match n with
    | 0 -> true
    | 1 -> false
    | n -> even (n-2)
```

```
for all n:int,
  even (2 * n) == true
```

But can we do proofs about entire *programming languages*?

In other words, proofs about *all programs that anyone could ever write in the programming language*?

But there are so many programs ... how do we even get started?

We often think about programs as if they are functions.

But is there another way to represent these functions?

A Trick

Consider assignment #4.

We are able to represent all programs using a data type:

```
type exp =
  Var of variable
| Const of constant
| Op of exp * op * exp
...
```

A Trick

Consider assignment #4.

We are able to represent all programs using a data type:

```
type exp =
   Var of variable
| Const of constant
| Op of exp * op * exp
...
```

We know how to prove things about functions over datatypes, so we know how to prove things about programming languages.

What Kinds of Things Might We Prove About PLs?

We typically prove things about functions over data types.

What kinds of functions over programs are there?

```
type exp =
  Var of variable
| Const of constant
| Op of exp * op * exp
...
```

What Kinds of Things Might We Prove About PLs?

We typically prove things about functions over data types.

What kinds of functions over programs are there?

```
type exp =
  Var of variable
| Const of constant
| Op of exp * op * exp
...
```

```
let eval (e:exp) = ...
let synthesize (s:spec) : exp = ...
let type_check (e:exp) = ...
let terminates (e:exp)  = ...
let closed (e:exp) = ...
let is_pure (e:exp) = ...
let compile (e:exp)  = ...
let optimize (e:exp) = ...
let is_correct (s:spec) (e:exp) = ...
let refactor (e:exp) = ...
```

ACM DL DIGITAL LIBRARY

Princeton University

SIGN IN SIGN UP

SEARCH

Conferences

POPL

POPL Principles of Programming Languages

The annual Symposium on Principles of Programming Languages is a forum for the discussion of all aspects of programming languages and systems, with emphasis on how principles underpin practice. Both theoretical and experimental papers are welcome, on topics ranging from formal frameworks to experience reports.

Search within POPL: SEARCH

About | Award Winners | Authors | Affiliations | Upcoming Conferences | Sponsors | Publication Archive | Web Archive

POPL subject areas

Compilers Formal language definitions Formal languages and automata theory Formal software verification Lambda calculus Language features Language types Logic Program reasoning Program semantics Program verification Semantics and reasoning Software development process management Type structures Verification

Bibliometrics: publication history

Publication years	1973-2018
Publication count	1,983
Citation Count	51,895
Available for download	1,829
Downloads (6 Weeks)	3,672
Downloads (12 Months)	42,478
Downloads (cumulative)	767,519
Average downloads per article	419.64
Average citations per article	26.17

PROOFS ABOUT PROGRAMMING LANGUAGES: AN EXAMPLE

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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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)

IH2: 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 =
match e with
  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)

IH2: 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 =
match e with
  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)   **IH2:** 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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 (Add(Int 0, e2))	(RHS)
== eval (opt e2)	(by eval opt)	== (eval(Int 0)) + (eval e2)	(eval)
== eval e2	(by IH)	== 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 =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1
| 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 =
match e with
  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)

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 =
match e with
  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)

IH2: 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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  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 =
match e with
  Int i -> i
| Add (e1, e2) -> (eval env e1) + (e
| 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 s 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 =
match e with
  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 =
match e with
  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 =
match e with
  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 done!
(we showed the
LHS == RHS)

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 : e          -> int

let rec eval (env: e
match
  Int i ->
| Add (e1, e2
| Var x -> lookup
```

```
let rec opt (e:exp) : exp =
  Int i -> Int i
| Add (Int 0, e) -> opt e
| Add (e, Int 0) -> opt e
|             1,e2) ->
         (opt e1, opt e2)
       x -> Var x
```

al (opt e) == eval e

PROOF DONE!!!

Proof:

e!
the
LHS

Case: e = Var x

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

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: for all t: 'a tree, 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: for all t: ‘a tree, flip(flip t) = t.

Theorem: for all t: ‘a tree, flip(flip (flip t)) = flip t.