

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>

ASSIGNMENT NOTES

Assignment #4 is pretty long

If you haven't started – get started!

Last year, more than half of all late days were used on this assignment!

If you want feedback on your assignment before the midterm (which I think could be helpful), you must turn it in on time.

We can't promise to be able to grade the assignment before the midterm if you use late days and turn it in late.

A Note about Free Variables

What are the free variables of the following expression?

```
if true then
  x
else
  y
```

What are the free variables of the following expression?

```
(fun x y ->
  match x with
  [] -> 0
  | hd::tl -> w + hd) [] z
```

A Note about Free Variables

Key points:

- the free variables are determined by the syntactic structure (ie, by the tree-shape and the occurrence of binding sites) of the expression *only*
- the free variables are *independent of how an expression is evaluated.*

PROOFS

Equational Reasoning: Some Key Ideas

What is the fundamental *definition of expression equality* ($e_1 == e_2$)?

- 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 $e_1 \rightarrow e_2$ then $e_1 == e_2$
- if $e_1 == e_2$ then $e[e_1/x] == e[e_2/x]$. (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 ,

$$\text{add_all} (\text{add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

```
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 ,

$$\text{add_all} (\text{add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

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 ,

$$\text{add_all (add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = []$:

$$\begin{aligned} & \text{add_all (add_all } [] \ a) \ b && \text{(LHS of theorem)} \\ == & \end{aligned}$$

```
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 ,

$$\text{add_all} (\text{add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = []$:

$$\begin{aligned} & \text{add_all} (\text{add_all } [] \ a) \ b \\ == & \text{add_all } [] \ b \\ == & \end{aligned}$$

(LHS of theorem)

(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 ,

$$\text{add_all (add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = []$:

$\text{add_all (add_all [] } a) \ b$	(LHS of theorem)
$== \text{add_all [] } b$	(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 ,

$$\text{add_all} (\text{add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = []$:

$\text{add_all} (\text{add_all } [] \ a) \ b$	(LHS of theorem)
$== \text{add_all } [] \ b$	(by evaluation of add_all)
$== []$	(by evaluation of add_all)
$== \text{add_all } [] \ (a + b)$	(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 ,

$$\text{add_all (add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = hd :: tl$:

$$\begin{aligned} & \text{add_all (add_all (hd :: tl) a) b} && \text{(LHS of theorem)} \\ == & \end{aligned}$$

```
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 ,

$$\text{add_all} (\text{add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = hd :: tl$:

$$\begin{aligned} & \text{add_all} (\text{add_all} (hd :: tl) \ a) \ b \\ == & \text{add_all} ((hd+a) :: \text{add_all } tl \ a) \ b \\ == & \end{aligned}$$

(LHS of theorem)
(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 ,

$$\text{add_all (add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = hd :: tl$:

$\text{add_all (add_all (hd :: tl) a) b}$	(LHS of theorem)
$== \text{add_all ((hd+a) :: add_all tl a) b}$	(by eval inner add_all)
$== (hd+a+b) :: (\text{add_all (add_all tl a) b})$	(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 ,

$$\text{add_all} (\text{add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = hd :: tl$:

$\text{add_all} (\text{add_all} (hd :: tl) \ a) \ b$	(LHS of theorem)
$== \text{add_all} ((hd+a) :: \text{add_all } tl \ a) \ b$	(by eval inner <code>add_all</code>)
$== (hd+a+b) :: (\text{add_all} (\text{add_all } tl \ a) \ b)$	(by eval outer <code>add_all</code>)
$== (hd+a+b) :: \text{add_all } tl \ (a+b)$	(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 ,

$$\text{add_all (add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = hd :: tl$:

$\text{add_all (add_all (hd :: tl) a) b}$	(LHS of theorem)
$== \text{add_all ((hd+a) :: add_all tl a) b}$	(by eval inner add_all)
$== (hd+a+b) :: (\text{add_all (add_all tl a) b})$	(by eval outer add_all)
$== (hd+a+b) :: \text{add_all tl (a+b)}$	(by IH)
$== (hd+(a+b)) :: \text{add_all tl (a+b)}$	(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 ,

$$\text{add_all (add_all } xs \ a) \ b == \text{add_all } xs \ (a+b)$$

Proof: By induction on xs .

case $xs = hd :: tl$:

$\text{add_all (add_all (hd :: tl) a) b}$	(LHS of theorem)
$== \text{add_all ((hd+a) :: add_all tl a) b}$	(by eval inner add_all)
$== (hd+a+b) :: (\text{add_all (add_all tl a) b})$	(by eval outer add_all)
$== (hd+a+b) :: \text{add_all tl (a+b)}$	(by IH)
$== (hd+(a+b)) :: \text{add_all tl (a+b)}$	(associativity of +)
$== \text{add_all (hd::tl) (a+b)}$	(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 , $\text{property}(xs)$.

Proof: By induction on lists xs .

Case: $xs == []$:

... no uses of IH ...

Case: $xs == hd :: tl$:

IH: $\text{property}(tl)$

Template for Inductive Proofs on Lists

Theorem: For all lists xs , $\text{property}(xs)$.

Proof: By induction on lists xs .

Case: $xs == []$:

... no uses of IH ...

one case for empty list



Case: $xs == hd :: tl$:

IH: $\text{property}(tl)$

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 , $\text{property}(xs)$.

Proof: By induction on lists xs .

Case: $xs == []$:

... no uses of IH ...

one case for empty list

Case: $xs == hd :: tl$:

IH: $\text{property}(tl)$

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 tail of non-empty 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 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 $x : t$, $\text{property}(x)$.

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

More General Template for Inductive Datatypes

$\text{type } t = C1 \text{ of } t1 \mid C2 \text{ of } t2 \mid \dots \mid Cn \text{ of } tn$

Theorem: For all $x : t$, $\text{property}(x)$.

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 $f : b \rightarrow c$,
For all (total) functions $g : a \rightarrow b$,
For all trees $t : \text{a tree}$,
 $\text{tm } f (\text{tm } g t) == \text{tm } (f \langle \rangle g) t$

“Forall intro”

Theorem:

For all (total) functions $f : b \rightarrow c$,
For all (total) functions $g : a \rightarrow b$,
For all trees $t : \text{a tree}$,
 $\text{tm } f (\text{tm } g t) == \text{tm } (f \langle \rangle g) t$

```
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 $t : \text{a tree}$,
 $\text{tm } f (\text{tm } g t) == \text{tm } (f \langle \rangle g) t$

Another example

Theorem:

For all trees t : a tree,
 $\text{tm } f (\text{tm } g \ t) == \text{tm } (f \langle \rangle g) \ t$

```
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,
 $\text{tm } f (\text{tm } g \ t) == \text{tm } (f \langle \rangle g) \ t$

```
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: $t = \text{Leaf}$

No inductive hypothesis to use.

(Leaf doesn't contain any smaller components with type tree.)

Proof:

$\text{tm } f (\text{tm } g \ \text{Leaf})$

Another example

Theorem:

For all trees t : a tree,
 $\text{tm } f (\text{tm } g t) == \text{tm } (f \langle \rangle g) t$

```
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: $t = \text{Leaf}$

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,

$\text{tm } f (\text{tm } g \ t) == \text{tm } (f \langle \rangle g) \ t$

Case: $t = \text{Node}(v, l, r)$

IH1: $\text{tm } f (\text{tm } g \ l) == \text{tm } (f \langle \rangle g) \ l$

IH2: $\text{tm } f (\text{tm } g \ r) == \text{tm } (f \langle \rangle g) \ r$

```
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,
 $\text{tm } f (\text{tm } g t) == \text{tm } (f \langle \rangle g) t$

Case: $t = \text{Node}(v, l, r)$

IH1: $\text{tm } f (\text{tm } g l) == \text{tm } (f \langle \rangle g) l$

IH2: $\text{tm } f (\text{tm } g r) == \text{tm } (f \langle \rangle g) r$

Proof:

$\text{tm } f (\text{tm } g (\text{Node } (v, l, r)))$

$== \text{tm } (f \langle \rangle g) (\text{Node } (v, l, r))$

```
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,

$$\text{tm } f (\text{tm } g \ t) == \text{tm } (f \langle \rangle g) \ t$$

Case: $t = \text{Node}(v, l, r)$

$$\text{IH1: } \text{tm } f (\text{tm } g \ l) == \text{tm } (f \langle \rangle g) \ l$$

$$\text{IH2: } \text{tm } f (\text{tm } g \ r) == \text{tm } (f \langle \rangle g) \ r$$

Proof:

$$\begin{aligned} & \text{tm } f (\text{tm } g (\text{Node } (v, l, r))) \\ == & \text{tm } f (\text{Node } (g \ v, \text{tm } g \ l, \text{tm } g \ r)) \end{aligned}$$

(eval inner tm)

$$== \text{tm } (f \langle \rangle g) (\text{Node } (v, l, r))$$

```
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,
 $\text{tm } f (\text{tm } g t) == \text{tm } (f \langle \rangle g) t$

Case: $t = \text{Node}(v, l, r)$

IH1: $\text{tm } f (\text{tm } g l) == \text{tm } (f \langle \rangle g) l$

IH2: $\text{tm } f (\text{tm } g r) == \text{tm } (f \langle \rangle g) r$

Proof:

$\text{tm } f (\text{tm } g (\text{Node } (v, l, r)))$
 $== \text{tm } f (\text{Node } (g v, \text{tm } g l, \text{tm } g r))$

(eval inner tm)

$\text{Node } ((f \langle \rangle g) v, \text{tm } (f \langle \rangle g) l, \text{tm } (f \langle \rangle g) r)$
 $== \text{tm } (f \langle \rangle g) (\text{Node } (v, l, r))$

(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

Theorem:

For all trees t : a tree,
 $\text{tm } f (\text{tm } g t) == \text{tm } (f \langle \rangle g) t$

Case: $t = \text{Node}(v, l, r)$

IH1: $\text{tm } f (\text{tm } g l) == \text{tm } (f \langle \rangle g) l$

IH2: $\text{tm } f (\text{tm } g r) == \text{tm } (f \langle \rangle g) r$

Proof:

$\text{tm } f (\text{tm } g (\text{Node } (v, l, r)))$
 $== \text{tm } f (\text{Node } (g v, \text{tm } g l, \text{tm } g r))$ (eval inner tm)
 $== \text{Node } (f (g v), \text{tm } f (\text{tm } g l), \text{tm } f (\text{tm } g r))$ (eval – since g, tm are total)

$\text{Node } ((f \langle \rangle g) v, \text{tm } (f \langle \rangle g) l, \text{tm } (f \langle \rangle g) r)$
 $== \text{tm } (f \langle \rangle g) (\text{Node } (v, l, r))$ (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

Theorem:

For all trees t : a tree,

$$\text{tm } f (\text{tm } g \ t) == \text{tm } (f \langle \rangle g) \ t$$

Case: $t = \text{Node}(v, l, r)$

$$\text{IH1: } \text{tm } f (\text{tm } g \ l) == \text{tm } (f \langle \rangle g) \ l$$

$$\text{IH2: } \text{tm } f (\text{tm } g \ r) == \text{tm } (f \langle \rangle g) \ r$$

Proof:

$$\begin{aligned} & \text{tm } f (\text{tm } g (\text{Node } (v, l, r))) \\ == & \text{tm } f (\text{Node } (g \ v, \text{tm } g \ l, \text{tm } g \ r)) && \text{(eval inner tm)} \\ == & \text{Node } (f (g \ v), \text{tm } f (\text{tm } g \ l), \text{tm } f (\text{tm } g \ r)) && \text{(eval – since } g, \text{tm are total)} \\ \\ & \text{Node } ((f \langle \rangle g) \ v, \text{tm } (f \langle \rangle g) \ l, \text{tm } f (\text{tm } g \ r)) \\ == & \text{Node } ((f \langle \rangle g) \ v, \text{tm } (f \langle \rangle g) \ l, \text{tm } (f \langle \rangle g) \ r) && \text{(IH2)} \\ == & \text{tm } (f \langle \rangle g) (\text{Node } (v, l, r)) && \text{(eval reverse)} \end{aligned}$$

```
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,

$$\text{tm } f (\text{tm } g \ t) == \text{tm } (f \langle \rangle g) \ t$$

Case: $t = \text{Node}(v, l, r)$

$$\text{IH1: } \text{tm } f (\text{tm } g \ l) == \text{tm } (f \langle \rangle g) \ l$$

$$\text{IH2: } \text{tm } f (\text{tm } g \ r) == \text{tm } (f \langle \rangle g) \ r$$

Proof:

$$\begin{aligned} & \text{tm } f (\text{tm } g (\text{Node } (v, l, r))) \\ == & \text{tm } f (\text{Node } (g \ v, \text{tm } g \ l, \text{tm } g \ r)) && \text{(eval inner tm)} \\ == & \text{Node } (f (g \ v), \text{tm } f (\text{tm } g \ l), \text{tm } f (\text{tm } g \ r)) && \text{(eval – since g, tm are total)} \\ == & \text{Node } ((f \langle \rangle g) \ v, \text{tm } f (\text{tm } g \ l), \text{tm } f (\text{tm } g \ r)) \\ == & \text{Node } ((f \langle \rangle g) \ v, \text{tm } (f \langle \rangle g) \ l, \text{tm } f (\text{tm } g \ r)) && \text{(IH1)} \\ == & \text{Node } ((f \langle \rangle g) \ v, \text{tm } (f \langle \rangle g) \ l, \text{tm } (f \langle \rangle g) \ r) && \text{(IH2)} \\ == & \text{tm } (f \langle \rangle g) (\text{Node } (v, l, r)) && \text{(eval reverse)} \end{aligned}$$

```
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,

$$\text{tm } f (\text{tm } g \ t) == \text{tm } (f \langle \rangle g) \ t$$

Case: $t = \text{Node}(v, l, r)$

$$\text{IH1: } \text{tm } f (\text{tm } g \ l) == \text{tm } (f \langle \rangle g) \ l$$

$$\text{IH2: } \text{tm } f (\text{tm } g \ r) == \text{tm } (f \langle \rangle g) \ r$$

Proof:

$$\begin{aligned} & \text{tm } f (\text{tm } g (\text{Node } (v, l, r))) \\ == & \text{tm } f (\text{Node } (g \ v, \text{tm } g \ l, \text{tm } g \ r)) && \text{(eval inner tm)} \\ == & \text{Node } (f \ (g \ v), \text{tm } f (\text{tm } g \ l), \text{tm } f (\text{tm } g \ r)) && \text{(eval – since } g, \text{tm are total)} \\ == & \text{Node } ((f \langle \rangle g) \ v, \text{tm } f (\text{tm } g \ l), \text{tm } f (\text{tm } g \ r)) && \text{(eval reverse)} \\ == & \text{Node } ((f \langle \rangle g) \ v, \text{tm } (f \langle \rangle g) \ l, \text{tm } f (\text{tm } g \ r)) && \text{(IH1)} \\ == & \text{Node } ((f \langle \rangle g) \ v, \text{tm } (f \langle \rangle g) \ l, \text{tm } (f \langle \rangle g) \ r) && \text{(IH2)} \\ == & \text{tm } (f \langle \rangle g) (\text{Node } (v, l, r)) && \text{(eval reverse)} \end{aligned}$$

```
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)
```

Summary: Proof Template for Trees

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

Theorem: For all $x : 'a \text{ tree}$, $\text{property}(x)$.

Proof: By induction on the structure of trees x .

Case: $x == \text{Leaf}$:

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

Case: $x == \text{Node}(v, \text{left}, \text{right})$:

IH1: $\text{property}(\text{left})$

IH2: $\text{property}(\text{right})$

... 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) = ...
```

Conferences



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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Int } i$

$\text{eval} (\text{opt} (\text{Int } i))$

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Int } i$

$\text{eval} (\text{opt} (\text{Int } i))$ (RHS)
 $== \text{eval} (\text{Int } i)$ (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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Int } i$

$\text{eval} (\text{opt} (\text{Int } i))$ (RHS)
 $== \text{eval} (\text{Int } i)$ (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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

IH: $\text{eval} (\text{opt } e2) == \text{eval } e2$

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

IH: $\text{eval} (\text{opt } e2) == \text{eval } e2$

$\text{eval} (\text{opt} (\text{Add}(\text{Int } 0, e2)))$ (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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

IH: $\text{eval} (\text{opt } e2) == \text{eval } e2$

$\text{eval} (\text{opt} (\text{Add}(\text{Int } 0, e2)))$ (LHS)
 $== \text{eval} (\text{opt } e2)$ (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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

IH: $\text{eval} (\text{opt } e2) == \text{eval } e2$

```
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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

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

$\text{eval} (\text{Add}(\text{Int } 0, e2))$ (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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

```
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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

```
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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

```
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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

$\text{eval} (\text{opt} (\text{Add}(\text{Int } 0, e2)))$ (LHS)
 $== \text{eval} (\text{opt } e2)$ (by eval opt)
 $== \text{eval } e2$ (by IH)

$\text{eval} (\text{Add}(\text{Int } 0, e2))$ (RHS)
 $== (\text{eval}(\text{Int } 0)) + (\text{eval } e2)$ (eval)
 $== 0 + \text{eval } e2$ (eval)
 $== \text{eval } e2$ (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:

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

$\text{eval } e, \text{eval } (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(\text{Int } 0, e2)$

$\text{eval } (\text{opt } (\text{Add}(\text{Int } 0, e2)))$ (LHS)
 $== \text{eval } (\text{opt } e2)$ (by eval opt)
 $== \text{eval } e2$ (by IH)

$\text{eval } (\text{Add}(\text{Int } 0, e2))$ (RHS)
 $== (\text{eval}(\text{Int } 0)) + (\text{eval } e2)$ (eval)
 $== 0 + \text{eval } e2$ (eval)
 $== \text{eval } e2$ (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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e2, \text{Int } 0)$

IH: $\text{eval} (\text{opt } e2) == \text{eval } e2$

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e2, \text{Int } 0)$

IH: $\text{eval} (\text{opt } e2) == \text{eval } e2$

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e1, e2)$

IH1: $\text{eval} (\text{opt } e1) == \text{eval } e1$

IH2: $\text{eval} (\text{opt } e2) == \text{eval } e2$

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e1, e2)$

$\text{eval} (\text{opt} (\text{Add}(e1, e2)))$ (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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e1, e2)$

$\text{eval} (\text{opt} (\text{Add}(e1, e2)))$ (LHS)
 $== \text{eval} (\text{Add} (\text{opt } e1, \text{opt } e2))$ (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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e1, e2)$

```
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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e1, e2)$

```
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)
```

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e1, e2)$

```
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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Add}(e1, e2)$

```
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
```

case done!
(we showed the LHS == RHS) $(opt\ e) == eval\ e$

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

Case: $e = Add(e1, e2)$

$eval\ (opt\ (Add(e1, e2)))$ (LHS)	$eval\ (Add(e1, e2))$ (RHS)
$== eval\ (Add\ (opt\ e1, opt\ e2))$ (by eval opt)	$== (eval\ e1) + (eval\ e2)$ (eval)
$== eval\ (opt\ e1) + eval\ (opt\ e2)$ (by eval 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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Var } x$

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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of expressions $e : \text{exp}$.

Case: $e = \text{Var } x$

```
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 $e : \text{exp}$, $\text{eval} (\text{opt } e) == \text{eval } e$

Proof: By induction on the structure of e

Case: $e = \text{Var } x$

$\text{eval} (\text{opt} (\text{Var } x))$ (LHS)
 $== \text{eval} (\text{Var } x)$ (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 id
```

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

```
let rec eval (env: env)
```

```
match e with
| 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
  | Add (e1, e2) ->
    (opt e1, opt e2)
  | Var x -> Var x
```

PROOF DONE!!!

eval (opt e) == eval e

Proof: P

Case: e = Var x

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

LHS

e!
the

Summary of Template for Inductive Datatypes

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

Theorem: For all $x : t$, $\text{property}(x)$.

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 ...

use patterns
that divide
up the cases

Take inspiration
from the
structure of the
program

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, $\text{flip}(\text{flip } t) = t$.

Theorem: for all t: 'a tree, $\text{flip}(\text{flip}(\text{flip } t)) = \text{flip } t$.