More Proofs By Induction
(Trees and General Datatypes)

COS 326
Andrew W. Appel
Princeton University

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 \( xs \),

\[
\text{add\_all } (\text{add\_all } xs \ a) \ b \ = \ = \ \text{add\_all } xs \ (a+b)
\]

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 \( xs \),

\[
\text{add\_all}\ (\text{add\_all}\ xs\ a)\ b\ ==\ \text{add\_all}\ xs\ (a+b)
\]

**Proof:** By induction on \( xs \).

\[
\text{case}\ xs\ =\ [\ ]:\
\]

\[
\text{add\_all}\ (\text{add\_all}\ [\ ]\ a)\ b\ \ \ (\text{LHS of theorem})
\]

\[
==
\]

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

**Theorem:** For all lists $xs$,

$$\text{add\_all (add\_all \, xs \, a) \, b \, == \, add\_all \, xs \, (a+b)}$$

**Proof:** By induction on $xs$.

case $xs = [\,]$:

- $\text{add\_all (add\_all \, [] \, a) \, b}$
- $\, == \, \text{add\_all \, [] \, b}$
- $\, ==$ (LHS of theorem)
- (by evaluation of add\_all)

```ocaml
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 \(xs\).

\[
\text{case } xs = [ ]:
\]
\[
\text{add\_all}\ (\text{add\_all}\ [ ]\ a)\ b\ =\ =\ \text{add\_all}\ [ ]\ b\ =\ [ ]\ =\ 
\]

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

\[
\text{add\_all } (\text{add\_all } [ ] \ a) \ b \quad \text{(LHS of theorem)}
\]
\[
= \text{add\_all } [ ] \ b \quad \text{(by evaluation of add\_all)}
\]
\[
= [ ] \quad \text{(by evaluation of add\_all)}
\]
\[
= \text{add\_all } [ ] \ (a + b) \quad \text{(by evaluation of add\_all)}
\]

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 \(xs = \text{hd}::\text{tl}\):

\[
\text{add\_all}\ (\text{add\_all}\ (\text{hd}\::\text{tl})\ a)\ b\quad\text{(LHS of theorem)}
\]

==

let rec add\_all xs c =
  match xs with
  | [] -> []
  | hd::tl -> (hd+c)::add\_all tl c
Theorem: For all lists \( xs \),
\[
\text{add\_all}\ (\text{add\_all}\ xs\ a)\ b\ =\ =\ \text{add\_all}\ xs\ (a+b)
\]

Proof: By induction on \( xs \).

\[
\text{case}\ \text{x} = \text{hd} :: \text{tl}:
\]

\[
\text{add\_all}\ (\text{add\_all}\ (\text{hd} :: \text{tl})\ a)\ b \quad \text{(LHS of theorem)}
\]
\[
=\ \text{add\_all}\ ((\text{hd}+a) :: \text{add\_all}\ \text{tl}\ a)\ b \quad \text{(by eval inner add\_all)}
\]
\[
=\ 
\]

```
let rec add_all xs c =
  match xs with
  | [] -> []
  | hd::tl -> (hd+c)::add_all tl
```
Theorem: For all lists \(xs\),

\[
\text{add\_all (add\_all \, xs \, a) \, b} = \text{add\_all \, xs \, (a+b)}
\]

Proof: By induction on \(xs\).

\[
\text{case } xs = \text{hd :: tl:}
\]

\[
\text{add\_all (add\_all (hd :: tl) \, a) \, b} \quad \text{(LHS of theorem)}
\]

\[
= \text{add\_all ((hd+a) :: add\_all \, tl \, a) \, b} \quad \text{(by eval inner add\_all)}
\]

\[
= (hd+a+b) :: (\text{add\_all (add\_all \, tl \, a) \, b}) \quad \text{(by eval outer add\_all)}
\]

\[
= \]

let rec add_all xs c =
match xs with
| [] -> []
| hd::tl -> (hd+c)::add_all tl c
Theorem: For all lists \( xs \),
\[
\text{add\_all } (\text{add\_all } xs\ a)\ b\ =\ =\ \text{add\_all } xs\ (a+b)
\]

Proof: By induction on \( xs \).

\[
\text{case } xs = hd :: tl:
\]

\[
\text{add\_all } (\text{add\_all } (hd :: tl)\ a)\ b\ =\ =\ \text{add\_all } ((hd+a) :: \text{add\_all } tl\ a)\ b\ =\ (by\ eval\ inner\ add\_all)
\]
\[
\text{add\_all } ((hd+a+b) :: (\text{add\_all } (\text{add\_all } tl\ a)\ b)\ =\ (by\ eval\ outer\ add\_all)
\]
\[
\text{add\_all } ((hd+a+b) :: \text{add\_all } tl\ (a+b)\ =\ (by\ IH)
\]

let rec add_all xs c =
  match xs with
  | [] -> []
  | hd::tl -> (hd+c)::add_all tl c
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 $xs = \text{hd} :: \text{tl}$:

\[
\begin{align*}
\text{add\_all} (\text{add\_all} (\text{hd} :: \text{tl}) \; a) \; b &\quad \quad \text{(LHS of theorem)} \\
= &\quad \quad \text{(by eval inner add\_all)} \\
= &\quad \quad \text{(by eval outer add\_all)} \\
= &\quad \quad \text{(by IH)} \\
= &\quad \quad \text{(associativity of + )}
\end{align*}
\]

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

**Theorem:** For all lists $xs$,

$$
\text{add\_all}\ (\text{add\_all}\ xs\ a)\ b \equiv \text{add\_all}\ xs\ (a+b)
$$

**Proof:** By induction on $xs$.

\[
\text{case } xs = \text{hd :: tl}:
\]

\[
\begin{align*}
\text{add\_all}\ (\text{add\_all}\ (\text{hd :: tl})\ a)\ b & \quad \text{(LHS of theorem)} \\
\equiv \text{add\_all}\ ((\text{hd}+a) :: \text{add\_all}\ tl\ a)\ b & \quad \text{(by eval inner add\_all)} \\
\equiv (\text{hd}+a+b) :: (\text{add\_all}\ (\text{add\_all}\ tl\ a)\ b) & \quad \text{(by eval outer add\_all)} \\
\equiv (\text{hd}+a+b) :: \text{add\_all}\ tl\ (a+b) & \quad \text{(by IH)} \\
\equiv (\text{hd}+(a+b)) :: \text{add\_all}\ tl\ (a+b) & \quad \text{(associativity of + )} \\
\equiv \text{add\_all}\ (\text{hd}::tl)\ (a+b) & \quad \text{(by (reverse) eval of add\_all)}
\end{align*}
\]

let rec add_all xs c = 
  match xs with 
  | [] -> []
  | hd::tl -> (hd+c)::add_all tl c
Theorem: For all lists \(xs\), \(\text{property}(xs)\).

Proof: By induction on lists \(xs\).

Case: \(xs == []\):
... no uses of IH ...

Case: \(xs == \text{hd} :: \text{tl}\):
IH: \(\text{property}(\text{tl})\)
Theorem: For all lists $xs$, property($xs$).
Proof: By induction on lists $xs$.

Case: $xs = []$:
... no uses of IH ...

Case: $xs = \text{hd} :: \text{tl}$:
IH: property($\text{tl}$)

one case for empty list

In general, cases must cover all the lists:
- other possibilities: case for $[]$, case for $x1::[]$, case for $x1::x2::tl$
**Theorem:** For all lists $xs$, $\text{property}(xs)$.

**Proof:** By induction on lists $xs$.

---

**Case:** $xs == [ ]$:

... no uses of $\text{IH}$ ...

---

**Case:** $xs == \text{hd} :: \text{tl}$:

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

---

one case for empty list

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

\[
\text{type } t = \text{ C1 of } t1 \mid \text{ C2 of } t2 \mid \ldots \mid \text{ Cn of } tn
\]

types \(t1, t2 \ldots tn\), may contain 1 or more occurrences of \(t\) within them.

Examples:

\[
\text{type mylist = MyNil} \\
| \text{ MyCons of int * mylist}
\]

\[
\text{type 'a tree = Leaf} \\
| \text{ Node of 'a * 'a tree * 'a tree}
\]

recursive occurrences
More General Template for Inductive Datatypes

\[
\text{type } t = \ C1 \text{ of } t1 \mid C2 \text{ of } t2 \mid \ldots \mid Cn \text{ of } tn
\]

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

**Proof:** By induction on structure of values \( x \) with type \( t \).
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

define a tree type as Leaf or Node of a * a tree * a tree

recursive definition of tm:
define tm function with argument f and t as:
  match t with
  | Leaf -> Leaf
  | Node (x, l, r) -> Node (f x, tm f l, tm f r)

pairing function <>:
define <> function with argument f and g as:
define 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
```
Theorem:
For all (total) functions $f : b \to c$, 
For all (total) functions $g : a \to b$, 
For all trees $t : a$ tree, 
$tm f (tm g t) = tm (f \langle> g) t$

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

let $\langle> f g =$ 
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, 
$tm f (tm g t) = tm (f \langle> g) t$
Theorem:
For all trees \( t : \text{ a tree} \),
\( \text{tm} \ f \ (\text{tm} \ g \ t) = \text{tm} \ (f <> g) \ t \)
Theorem:
For all trees $t : \text{a tree},$
\[ \text{tm } f (\text{tm } g \ t) == \text{tm } (f <> g) \ t \]

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

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,  
\[ \text{tm} f (\text{tm} g t) == \text{tm} (f <> g) t \]

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

IH1: \( \text{tm} f (\text{tm} g l) == \text{tm} (f <> g) l \)
IH2: \( \text{tm} f (\text{tm} g r) == \text{tm} (f <> g) r \)

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,  
\( \text{tm} \ f \ (\text{tm} \ g \ t) = \text{tm} \ (f \ <> \ g) \ t \)

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

\[ \begin{align*}  
\text{IH1:} \ & \text{tm} \ f \ (\text{tm} \ g \ l) = \text{tm} \ (f \ <> \ g) \ l \\
\text{IH2:} \ & \text{tm} \ f \ (\text{tm} \ g \ r) = \text{tm} \ (f \ <> \ g) \ r 
\end{align*} \]

Proof:  
\[ \text{tm} \ f \ (\text{tm} \ g \ (\text{Node} \ (v, l, r))) \]

== \( \text{tm} \ (f \ <> \ g) \ (\text{Node} \ (v, l, r)) \)
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,
\[
\text{tm } f \ (\text{tm } g \ t) = \text{tm } (f \ <> \ g) \ t
\]

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

IH1: \( \text{tm } f \ (\text{tm } g \ l) = \text{tm } (f \ <> \ g) \ l \)
IH2: \( \text{tm } f \ (\text{tm } g \ r) = \text{tm } (f \ <> \ g) \ r \)

Proof:
\[
\text{tm } f \ (\text{tm } g \ (\text{Node}(v, l, r)))
= \text{tm } f \ (\text{Node}(g \ v, \text{tm } g \ l, \text{tm } g \ r)) \quad (\text{eval inner tm})
\]

\[
\text{Node} \ ((f \ <> \ g) \ v, \text{tm } (f \ <> \ g) \ l, \text{tm } (f \ <> \ g) \ r)
= \text{tm } (f \ <> \ g) \ (\text{Node}(v, l, r)) \quad (\text{eval reverse})
\]

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 \( t : \text{ a tree} \),
\[ \text{tm } f \ (\text{tm } g \ t) = \text{tm } (f \ <> \ g) \ t \]

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

IH1: \( \text{tm } f \ (\text{tm } g \ l) = \text{tm } (f \ <> \ g) \ l \)
IH2: \( \text{tm } f \ (\text{tm } g \ r) = \text{tm } (f \ <> \ g) \ r \)

Proof:
\[
\begin{align*}
\text{tm } f \ (\text{tm } g \ (\text{Node}(v, l, r))) \\
= \text{tm } f \ (\text{Node}(g \ v, \text{tm } g \ l, \text{tm } g \ r)) \\
= \text{Node}(f \ (g \ v), \text{tm } f \ (\text{tm } g \ l), \text{tm } f \ (\text{tm } g \ r))
\end{align*}
\]

(eval inner tm)
(eval – since \( g, \text{tm} \) are total)

\[
\begin{align*}
\text{Node} \ ((f \ <> \ g) \ v, \ (f \ <> \ g) \ l, \ (f \ <> \ g) \ r) \\
= \text{tm } (f \ <> \ g) \ (\text{Node}(v, l, r))
\end{align*}
\]
(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,
\[
\text{tm} f \ (\text{tm} g \ t) = \text{tm} \ (f <\!>\! g) \ t
\]

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

**IH1:** \( \text{tm} f \ (\text{tm} g \ l) = \text{tm} \ (f <\!>\! g) \ l \)

**IH2:** \( \text{tm} f \ (\text{tm} g \ r) = \text{tm} \ (f <\!>\! g) \ r \)

**Proof:**
\[
\begin{align*}
\text{tm} f \ (\text{tm} g \ (\text{Node}(v, l, r))) \\
= \text{tm} f \ (\text{Node}(g v, \text{tm} g l, \text{tm} g r)) \\
= \text{Node}(f (g v), \text{tm} f \ (\text{tm} g l), \text{tm} f \ (\text{tm} g r)) \\
= \text{Node}((f <\!>\! g) v, \text{tm} ((f <\!>\! g) l, \text{tm} f \ (\text{tm} g r))) \\
= \text{Node}((f <\!>\! g) v, \text{tm} (f <\!>\! g) l, \text{tm} (f <\!>\! g) r) \\
= \text{tm} (f <\!>\! g) \ (\text{Node}(v, l, r))
\end{align*}
\]

\[
\text{let rec tm f t =} \\
\quad \text{match t with} \\
\quad \quad | \text{Leaf} -> \text{Leaf} \\
\quad \quad | \text{Node}(x, l, r) -> \text{Node}(f x, \text{tm} f l, \text{tm} f r)
\]

\[
\text{let } (<>f g =} \\
\quad \text{fun} \ x -> f (g x)
\]
Another example

Theorem:
For all trees \( t : \) a tree,
\[
\text{tm} f (\text{tm} g t) = \text{tm} (f \leftrightarrow g) t
\]

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

IH1: \( \text{tm} f (\text{tm} g l) = \text{tm} (f \leftrightarrow g) l \)
IH2: \( \text{tm} f (\text{tm} g r) = \text{tm} (f \leftrightarrow g) r \)

Proof:
\[
\begin{align*}
\text{tm} f (\text{tm} g (\text{Node} (v, l, r))) \\
&= \text{tm} f (\text{Node} (g v, \text{tm} g l, \text{tm} g r)) \\
&= \text{Node} (f (g v), \text{tm} f (\text{tm} g l), \text{tm} f (\text{tm} g r)) \\
&= \text{Node} ((f \leftrightarrow g) v, \text{tm} f (\text{tm} g l), \text{tm} f (\text{tm} g r)) \\
&= \text{Node} ((f \leftrightarrow g) v, \text{tm} (f \leftrightarrow g) l, \text{tm} f (\text{tm} g r)) \\
&= \text{Node} ((f \leftrightarrow g) v, \text{tm} (f \leftrightarrow g) l, \text{tm} (f \leftrightarrow g) r) \\
&= \text{tm} (f \leftrightarrow g) (\text{Node} (v, l, r))
\end{align*}
\]

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 \( t : \text{a tree}, \)
\( \text{tm} \ f \ (\text{tm} \ g \ t) \) \( = \) \( \text{tm} \ (f <> g) \ t \)

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

IH1: \( \text{tm} \ f \ (\text{tm} \ g \ l) \) \( = \) \( \text{tm} \ (f <> g) \ l \)
IH2: \( \text{tm} \ f \ (\text{tm} \ g \ r) \) \( = \) \( \text{tm} \ (f <> g) \ r \)

Proof:
\[
\begin{align*}
\text{tm} \ f \ (\text{tm} \ g \ (\text{Node} \ (v, l, r))) \\
&= \text{tm} \ f \ (\text{Node} \ (g \ v, \text{tm} \ g \ l, \text{tm} \ g \ r)) \\
&= \text{Node} \ (f \ (g \ v), \text{tm} \ f \ (\text{tm} \ g \ l), \text{tm} \ f \ (\text{tm} \ g \ r)) \\
&= \text{Node} \ ((f <> g) \ v, \text{tm} \ f \ (f <> g) \ l, \text{tm} \ f \ (f <> g) \ r) \\
&= \text{tm} \ (f <> g) \ (\text{Node} \ (v, l, r))
\end{align*}
\]
(eval inner \( \text{tm} \))
(eval – since \( g, \text{tm} \) are total)
(eval reverse)
(IH1)
(IH2)
(eval reverse)

let rec \( \text{tm} \ f \ t = \)
match \( t \) with
| Leaf \( \rightarrow \) Leaf
| Node \((x, l, r)\) \( \rightarrow \) Node \((f \ x, \text{tm} \ f \ l, \text{tm} \ f \ r)\)

let \((<>\)) \ f \ g =
fun \( x \rightarrow f \ (g \ x)\)
Theorem: For all $x : \text{'a tree}$, 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: $x == \text{Node (v, left, right)}$:

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

... use IH1 and IH2 in your proof ...
PROOFS ABOUT

PROGRAMMING LANGUAGES
You might wonder

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

```ocaml
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:

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

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?*

```plaintext
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?*

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

POPL subject areas

- Compilers
- Formal languages
- 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): 757,519
- Average downloads per article: 419.64
- Average citations per article: 26.17
PROOFS ABOUT PROGRAMMING LANGUAGES: AN EXAMPLE
A simple expression language

```plaintext
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

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

define type env
define 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

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
```

```
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
```

**Theorem:**
For all e : exp, eval (opt e) == eval e
A simple optimizer

```
type id = string

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

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

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

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

Case: e = Int i

  eval (opt (Int i))
A simple optimizer

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

\textbf{Case:} \( e = \text{Int} \ i \)

\[
\text{eval (opt (Int i)) (RHS)} \quad \text{==} \quad \text{eval (Int i)} \quad \text{(eval of opt)}
\]
A simple optimizer

**Type Definitions:**
- `type id = string`
- `type exp = Int of int | Add of exp * exp | Var of id`

**Function Definitions:**
- `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`

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

**Case:** `e = Int i`

\[
\text{eval (opt (Int i)) (RHS)} = \text{eval (Int i) (eval of opt)}
\]

**Theorem:** For all `e : exp`, `eval (opt e) == eval e`
A simple optimizer

```ocaml
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 definitions:

```plaintext
type id = string

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

type env

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

Recursive definitions:

```plaintext
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
```

Recursive function for optimization:

```plaintext
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

\[ \text{eval (opt (Add(Int 0, e2)))} \]  (LHS)
A simple optimizer

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

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

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 Definitions**

```plaintext
type id = string

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

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

$$\text{eval}(\text{Add}(\text{Int } 0, e2)) \quad \text{(RHS)}$$

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

$$= \text{eval } (\text{opt } e2) \quad \text{(by eval opt)}$$

$$= \text{eval } e2 \quad \text{(by IH)}$$
Proof: By induction on the structure of expressions e : exp.

Case: e = Add(Int 0, e2)

\[
\begin{align*}
\text{eval (opt (Add(Int 0, e2)))} \quad & \text{(LHS)} \\
\text{== eval (opt e2)} \quad & \text{(by eval opt)} \\
\text{== eval e2} \quad & \text{(by IH)}
\end{align*}
\]

\[
\begin{align*}
\text{eval (Add(Int 0, e2)))} \quad & \text{(RHS)} \\
\text{== (eval(Int 0)) + (eval e2)} \quad & \text{(eval)}
\end{align*}
\]
A simple optimizer

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

Case: e = Add(Int 0, e2)

\[
\begin{align*}
\text{eval (opt (Add(Int 0, e2)))} & \quad \text{(LHS)} \\
\text{== eval (opt e2)} & \quad \text{(by eval opt)} \\
\text{== eval e2} & \quad \text{(by IH)}
\end{align*}
\]

\[
\begin{align*}
\text{eval (Add(Int 0, e2))} & \quad \text{(RHS)} \\
\text{== (eval(Int 0)) + (eval e2)} & \quad \text{(eval)} \\
\text{== 0 + eval e2} & \quad \text{(eval)}
\end{align*}
\]
A simple optimizer

**Type Definitions**

- `type id = string`
- `type exp = Int of int | Add of exp * exp | Var of id`
- `type env`
- `val lookup : env -> id -> int`

**Functions**

- `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`

**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(Int 0)) + (eval e2) (eval)
 eval (opt e2) (by eval opt) == 0 + eval e2 (eval)
 eval e2 (by IH) == 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)

\[
\begin{align*}
eval \left(\text{opt} \left(\text{Add} (\text{Int} 0, \text{e2})\right)\right) & \quad \text{(LHS)} \\
& \quad \stackrel{\text{by eval opt}}{=} \quad \text{eval} \left(\text{opt} \text{e2}\right) \\
& \quad \stackrel{\text{by IH}}{=} \quad \text{eval} \text{e2} \\
\end{align*}
\]

\[
\begin{align*}
eval \left(\text{Add} (\text{Int} 0, \text{e2})\right) & \quad \text{(RHS)} \\
& \quad \stackrel{\text{eval}}{=} \quad (\text{eval} (\text{Int} 0)) + (\text{eval} \text{e2}) \\
& \quad \stackrel{\text{eval}}{=} \quad 0 + \text{eval} \text{e2} \\
& \quad \stackrel{\text{math}}{=} \quad \text{eval} \text{e2} \\
\end{align*}
\]
A simple optimizer

```ocaml
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
```

```ocaml
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
```

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 e2 (by IH)
```

```
eval (Add(Int 0, e2)) == (eval(Int 0)) + (eval e2) (eval)
== 0 + eval e2 (math)
== eval e2
```
A simple optimizer

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

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 definitions:**

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

**Functions:**

- `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)

\[
\text{eval (opt (Add(e1, e2)))} \quad \text{(LHS)}
\]
\[
\text{== eval (Add (opt e1, opt e2))} \quad \text{(by eval opt)}
\]
A simple optimizer

```ocaml
type id = string

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
```

**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 (opt (Add(e1, e2))) (LHS)
== eval (Add (opt e1, opt e2)) (by eval opt)
== eval (opt e1) + eval (opt e2) (by eval eval) (RHS)
A simple optimizer

type id = string
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)

\[
\begin{align*}
\text{eval (opt (Add(e1, e2)))} & \quad \text{(LHS)} \\
\text{== eval (Add (opt e1, opt e2))} & \quad \text{(by eval opt)} \\
\text{== eval (opt e1) + eval (opt e2)} & \quad \text{(by eval eval)}
\end{align*}
\]

\[
\begin{align*}
\text{eval (Add(e1, e2))} & \quad \text{(RHS)} \\
\text{== (eval e1) + (eval e2)} & \quad \text{(eval)}
\end{align*}
\]
A simple optimizer

**Type Definition**: 

- `type id = string`
- `type exp = Int of int | Add of exp * exp | Var of id`
- `type env`
- `val lookup : env -> id -> int`

**Evaluator**:

``` ocaml
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
```

**Optimizer**:

``` ocaml
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)`

1. `eval (opt (Add(e1, e2)))` (LHS)
2. `== eval (Add (opt e1, opt e2))` (by `eval opt`)
3. `== eval (opt e1) + eval (opt e2)` (by `eval eval`)
4. `eval (Add(e1, e2))` (RHS)
5. `== (eval e1) + (eval e2)` (eval)
6. `== 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 = Add(e1, e2)

\[
\begin{align*}
\text{eval (opt (Add(e1, e2)))} & \quad \text{(LHS)} \\
\text{== eval (Add (opt e1, opt e2))} & \quad \text{(by eval opt)} \\
\text{== eval (opt e1) + eval (opt e2)} & \quad \text{(by eval eval)}
\end{align*}
\]

\[
\begin{align*}
\text{eval (Add(e1, e2))} & \quad \text{(RHS)} \\
\text{== (eval e1) + (eval e2)} & \quad \text{(eval)} \\
\text{== eval (opt e1) + eval (opt e2)} & \quad \text{(by IH1 and IH2)}
\end{align*}
\]

**case done!**
(we showed the LHS == RHS)
A simple optimizer

```
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!

```
type id = string

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
```

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

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

```
A simple optimizer

**Type definitions:**
- `type id = string`
- `type exp = Int of int | Add of exp * exp | Var of id`
- `type env`
- `val lookup : env -> id -> int`

**Implementation:**
- `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`

**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

```ocaml
type id = string
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

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
```

**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)
```
A simple optimizer

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

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

**Case:** \( e = \text{Var} \ x \)

\[ \text{eval (opt (Var x))} = \text{eval (Var x)} \] (LHS)

by \( \text{eval opt} \)
Summary of Template for Inductive Datatypes

**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 \( t \) ...

**Type** \( t = C1 \ of \ t1 \ | \ C2 \ of \ t2 \ | ... | Cn \ of \ tn \)

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

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

let rec flip (t: 'a tree) =
  match t with
  | Leaf _ -> t
  | 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
  | 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.