More Proofs By Induction (Trees and General Datatypes)

COS 326

David Walker

Princeton University

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

slides copyright 2026 David Walker
permission granted to reuse these slides for non-commercial educational purposes

Another List example

Theorem: For all lists xs,

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

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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.

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)

=

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)
=	

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)
=	

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)

=

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)
=	

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)
=	

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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 +)

```
def add_all (xs : List Int) (c : Int) : List Int :=
  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)

```
def add_all (xs : List Int) (c : Int) : List Int :=
  match xs with
  | [ ] => [ ]
  | hd::tl => (hd+c)::add_all tl c
```

FORMALIZING THE EXAMPLE IN LEAN

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

```
def foo (xs : List Int) (c : Int) : List Int :=
  match xs with
    | [ ] => …
    | [hd] => …
    | hd::tl => …
```

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

Proof: By induction on lists xs.

Case: xs = [ ]:

… no uses of IH …

Case: xs = [hd]:

… no uses of IH …

Case: xs = hd :: tl:

IH: property(tl)

A function break down a list (or natural number) in a non-standard way.

- the breakdown must cover all cases (Lean or OCaml will check this)
- the cases for your inductive proof can follow the breakdown in the function

Template for Inductive Proofs on Lists

```
def foo (xs : List Int) (c : Int) : List Int :=
  match xs with
    | [ ] => …
    | [hd] => …
    | hd::tl => …
```

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

Proof: By induction on lists xs.

Case: xs = [ ]:

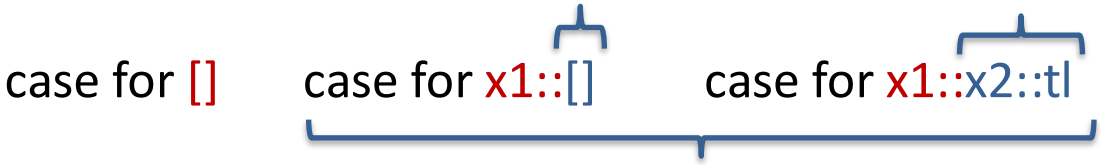
… no uses of IH …

Case: xs = [hd]:

… no uses of IH …

Case: xs = hd :: tl:

IH: property(tl)



This 3-way split is just a normal split into [] and x1::tl
Where there is a use of “cases” to split tl into [] and x2::tl’

Template for Inductive Proofs on Lists

```
def foo (xs : List Int) (c : Int) : List Int :=
  match xs with
    | [ ] => …
    | [hd] => …
    | hd::tl => …
```

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

Proof: By induction on lists xs.

Case: xs = [ ]:

… no uses of IH …

Case: xs = [hd]:

… no uses of IH …

Case: xs = hd :: tl:

IH: property(tl)

In Lean: there is a tactic called fun_induction, which helps you do the case split according to the cases that appear in a function.
It is not more powerful than ordinary induction, just more convenient.

POLYMORPHISM

One list type per element type?

Without polymorphism

```
inductive NList : Type where
  | nil
  | cons (n : Nat) (l : NList)

inductive BList : Type where
  | nil
  | cons (b : Bool) (l : BList)
```

...and a copy of `append`, `reverse`, `length` **and every lemma**, for every element type.

With polymorphism

```
inductive MyList (α : Type) : Type where
  | nil : MyList α
  | cons (x : α) (l : MyList α) : MyList α
```

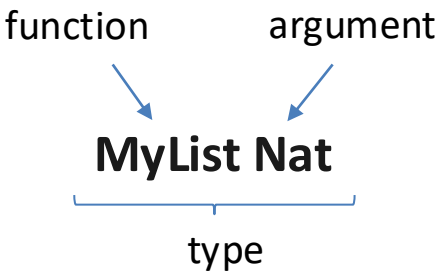
- α is a **type parameter**
- From this definition, we can create many types:
  - `MyList Nat`, `MyList Bool`, …
  - `MyList (MyList Nat)`

What is MyList?

```
inductive MyList (α : Type) : Type where
  | nil : MyList α
  | cons (x : α) (l : MyList α) : MyList α
```

MyList is a **function from types to types**.

Mylist : Type → Type



You ask Lean	Lean answers
`#check (MyList)`	`MyList : Type → Type`

What are nil and cons?

```
inductive MyList (α : Type) : Type where
  | nil : MyList α
  | cons (x : α) (l : MyList α) : MyList α
```

You ask Lean	Lean answers
`#check (MyList)`	`MyList : Type → Type`
`#check MyList.nil`	`MyList.nil {α : Type} : MyList α`

`#check MyList.cons`	`MyList.cons {α : Type}`
	`(x : α)`
	`(l : MyList α) : MyList α`

nil is a function that takes a type argument and produces a value

cons is a function that takes a type argument and 2 value args, and produces a value

What are nil and cons?

```
inductive MyList (α : Type) : Type where
  | nil : MyList α
  | cons (x : α) (l : MyList α) : MyList α
```

You ask Lean	Lean answers
`#check (MyList)`	`MyList : Type → Type`
`#check MyList.nil`	`MyList.nil {α : Type} : MyList α`

`#check MyList.cons`	`MyList.cons {α : Type}`
	`(x : α)`
	`(l : MyList α) : MyList α`

curly brackets indicate implicit arguments
that Lean tries to infer for you from context

if you supply 3 for x then
Lean knows α is Nat

What are nil and cons?

```
inductive MyList (α : Type) : Type where
  | nil : MyList α
  | cons (x : α) (l : MyList α) : MyList α
```

You ask Lean	Lean answers
`#check (MyList)`	`MyList : Type → Type`
`#check MyList.nil`	`MyList.nil {α : Type} : MyList α`

`#check MyList.cons`	`MyList.cons {α : Type}`
	`(x : α)`
	`(l : MyList α) : MyList α`

nil is a function that takes a type argument and produces a value

cons is a function that takes a type argument and 2 value args, and produces a value

What are nil and cons?

```
inductive MyList (α : Type) : Type where
  | nil : MyList α
  | cons (x : α) (l : MyList α) : MyList α
```

You ask Lean	Lean answers
`#check (MyList)`	`MyList : Type → Type`
`#check MyList.nil`	`MyList.nil {α : Type} : MyList α`

`#check MyList.cons`	`MyList.cons {α : Type}`
	`(x : α)`
	`(l : MyList α)`
	`: MyList α`

`#check MyList.cons 3` `MyList.nil`	`MyList.cons 3 MyList.nil : MyList Nat`

A polymorphic function: rep

```
def rep (α : Type) (x : α) (count : Nat) : MyList α :=
  match count with
  | 0 => .nil
  | count' + 1 => .cons x (rep α x count')
```

Apply `rep` to a type first, then to an element of that type (and a count):

```
example : rep Nat 4 2 = .cons 4 (.cons 4 .nil) := by rfl
example : rep Bool false 1 = .cons false .nil := by rfl
```

Type annotation inference

```
def rep' α (x : α) (count : Nat) : List α :=
  match count with
  | 0 => .nil
  | count' + 1 => .cons x (rep' α x count')
```

No type is written for α, yet Lean accepts it:

```
#check rep'
```

```
rep'.{u_1} (α : Type u_1) (x : α) (count : Nat) : List α
```

- **Type inference:** α is the argument of `List`, and `List` expects a type, so `α : Type`.
- `.{u_1}` and `Type u_1` are Lean's bookkeeping for treating types more generally.
  - Ignore them for now.
- Explicit annotations are still useful documentation, so we usually keep them.
- From here on we use Lean's built-in `List α : []` for nil, `x :: xs` for cons.

Type argument synthesis: holes

```
def rep'' (α : Type) (x : α) (count : Nat) : List α :=
  match count with
  | 0 => []
  | count' + 1 => x :: rep'' _ x count'
```

- A **hole** _ means: Lean, please work out what belongs here.

How Lean fills the hole

1. rep'' has type
   (α : Type) → α → Nat → List α

2. substitute _ for α:
   rep'' _ : _ → Nat → List _

3. (rep'' _) x must type check so arg x must be the same as _

4. So _ must be α.

Implicit arguments

```
def rep''' {α : Type} (x : α) (count : Nat) : List α :=
  match count with
  | 0 => []
  | count' + 1 => x :: rep''' x count'
```

- Curly braces make α implicit: callers never supply that argument.
- Lean fills the argument at every use, as if each call had _
- The recursive call leaves it out: `rep''' x count'`.

```
#eval rep''' 4 2                    -- [4, 4]
```

Implicit arguments

```
def rep''' {α : Type} (x : α) (count : Nat) : List α :=
  match count with
  | 0 => []
  | count' + 1 => x :: rep''' x count'
```

Supplying the type is an error, because Lean is not expecting it:

```
  | count' + 1 => x :: rep''' _ x count'
```

```
Application type mismatch: The argument x has type α
but is expected to have type Nat in the application rep'''
?m.13 x
```

Lean reads `_` as the element and `x` as the count.

Supplying type arguments explicitly

Lean cannot infer a type when it does not have enough information:

```
def mynil := List.nil
```

```
don't know how to synthesize implicit argument `α`
  @List.nil ?m.3
```

In such cases, use "@", which tells Lean to make all implicit arguments explicit:

```
def mynil := @List.nil Nat
```

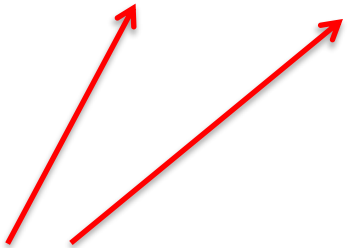
Alternately, use a type annotation:

```
def mynil : List Nat := List.nil
```

PROOFS ABOUT DATATYPES

More General Template for Inductive Datatypes

```
inductive t | C1 (v : t1) | C2 (v : t2) | ... | Cn (v : tn)
```



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

Examples:

```
inductive MyList where
  | MyNil
  | MyCons (n : Int) (l : MyList)
```

```
inductive Tree (α : Type) where
  | leaf
  | node (x : α) (l : Tree α) (r : Tree α)
```



recursive occurrences

More General Template for Inductive Datatypes

inductive t | C1 (v : t1) | C2 (v : t2) | … | Cn (v : 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

inductive t | C1 (v : t1) | C2 (v : t2) | ... | Cn (v : 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

```
inductive Tree (α : Type) | leaf | node (x : α) (l r : Tree α)

def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Another example

```
inductive Tree (α : Type) | leaf | node (x : α) (l r : Tree α)

def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Theorem:

For all (total) functions f : β → γ,
For all (total) functions g : α → β,
For all trees t : Tree α,
tm f (tm g t) = tm (f ∘ g) t

"Forall intro"

Theorem:
For all (total) functions f : β → γ,
For all (total) functions g : α → β,
For all trees t : Tree α,
tm f (tm g t) = tm (f ∘ g) t

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

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 : Tree α,
tm f (tm g t) = tm (f ∘ g) t

Another example

Theorem:
For all trees t : Tree α,
tm f (tm g t) = tm (f ∘ g) t

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Another example

Theorem:
For all trees t : Tree α,
tm f (tm g t) = tm (f ∘ g) t

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Case: t = leaf

No inductive hypothesis to use.
(leaf doesn't contain any smaller components with type Tree α.)

Proof:
tm f (tm g leaf)

Another example

Theorem:
For all trees t : Tree α,
tm f (tm g t) = tm (f ∘ g) t

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Case: t = leaf

No inductive hypothesis to use.
(leaf doesn't contain any smaller components with type Tree α.)

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 : Tree α,
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

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Another example

Theorem:
For all trees t : Tree α,
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)

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Another example

Theorem:
For all trees t : Tree α,
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)

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Another example

Theorem:
For all trees t : Tree α,
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

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Proof:
tm f (tm g (node v l r))
= tm f (node (g v) (tm g l) (tm g r)) (eval inner tm)

node ((f ∘ g) v) (tm (f ∘ g) l) (tm (f ∘ g) r)
= tm (f ∘ g) (node v l r) (eval reverse)

Another example

Theorem:
For all trees t : Tree α,
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

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

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 : Tree α,
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

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

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 : Tree α,
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

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

Proof:

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

Another example

Theorem:
For all trees t : Tree α,
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

```
def tm (f : α → β) (t : Tree α) : Tree β :=
  match t with
    | .leaf => .leaf
    | .node x l r => .node (f x) (tm f l) (tm f r)

def comp (f : β → γ) (g : α → β) : α → γ :=
  fun x => f (g x)    -- f ∘ g
```

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

inductive Tree (α : Type) | leaf | node (x : α) (l r : Tree α)

Theorem: For all x : Tree α, 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 ...