Inductive Proofs

COS 326

David Walker

Princeton University

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

COURSE STRUCTURE
PROJECTS

	Date	Events	Your Job
this week	Nov 3	Inductive Proofs	Think about project
	Nov 5	Inductive Proofs	Learn proof techniques
next week	Nov 10	Proof Systems and Logic	Assignment on proofs Proposal
	Nov 12	Proof Systems and Logic	
following week	Nov 17	Theorem Proving Assignment Due 1-Page Project Proposal Due	
	Nov 18-20	Debriefs: Theorem Proving + Project	
Project Work	Nov 24	Midterm 2	
	Dec 3-4	5-minute “One Cool Idea” Talks in Precept	
	Dec 15	Project Due; Dec 16-18 Debriefs	

Project Ideas

Build a new programming language:
Use infrastructure provided by assignment 4, but extend it substantially with interesting features drawn from OCaml, or Python or R or …

Build a SAT/SMT-based app:
Use infrastructure provided by assignment 5, but some new app.
Consider a game or scheduling problem or a synthesis problem

Prove your favorite data structure is correct:
Implement your favorite COS 226 data structure in Lean
(Or draw from any other class)
Specify appropriate correctness properties
Prove it correct

Pick your favorite theory class or math class. Could be high school.
Formalize a theorem in Lean. Prove it correct.
I am curious if Lean could be helpful in COS 240, 330, 445, etc.
Experiment for us!

Pick a different functional language or domain-specific language
Implement something cool. Jax. Dex. Integer linear solvers in python.

Some Past Projects (note: no Lean/SMT last year)

Functional Image Generation Library:
Build a a functional image generation library. Shapes. Colors. Composition

SQL Interpreter: SQL is a functional language! Build an interpreter

A functional programming language implemented in Rust
Infrastructure for FP, but in a different language of your choice

Simulator for circuits in subset of Verilog
An interpreter for hardware circuits

Crosswords in OCaml
Many games that involve some search can be implemented using SMT

CamlAx: Lightweight machine learning library
Implement your favorite algorithms from other classes in OCaml

OCaml Implementation of Probabilistic Programming Language. Dice

Programming Language for Origami.
Define a series of folds. Check for feasibility.

Project Grading

Focus: Communicate clearly to us what ***you learned*** by doing this project and what you can now ***do yourself***

- Done during intermediate debrief and final debrief without LLMs

Non-Focus: Size of the artifacts generated by your project.

- You might generate a large application
- You might generate a small application

Question: Can I use an LLM pro account to generate an impressive artifact?

- Yes, but what did *you* learn by doing that?
- What will *you* take away from this course?
- Our goal: A paid LLM pro account and 1000 agents provides no advantage

Expected time investment:

- Each week, spend time similar to the time you would spend on assignments
- 5-6h/week? 20-25h total? Spread it out and think about it in in-between times

PROOF TACTICS SO FAR

Proof Tactics: Equational Rewriting

Equational Rewriting
Informally:

```
Theorem: e1 = e2

Proof:
  e1        (LHS)
= e1'       (by evaluation)
= e1''      (by math)
= e2        (by evaluation)
```

claims

justifications

Equational Rewriting
In Lean:

```
Theorem foo : e1 = e2 := by

                  | e1   = e2    (initial goal)
ev                | e1'  = e2    (rewritten goal)
rewrite [ax]      | e1'' = e2    (rewritten goal)
ev                | e2   = e2    (rewritten goal)
rfl
```

justifications

infoview pane (proof state)

Proof Tactics: Quantifiers

Universal Quantifiers Informally

```
Theorem: ∀x:t.P(x)

Proof:
Consider arbitrary x of type t,
  proof of P(x)
  ...
```

Universal Quantifiers in Lean:

```
Theorem foo : ∀x:t.P(x) := by

                        |  |- ∀x:t.P(x)

  intro x               |   x:t |- P(x)
  proof of P(x)
  ...
```

In infoview, the turnstyle |- separates ***hypotheses*** from ***goals***.

hyp, hyp, hyp |- goal

Hypotheses are things we have assumed and can use in proofs.
Goals are things we are required to prove.

x:t is a kind of hypothesis -- an assumption about an object having a type

Proof Tactics: Quantifiers

Universal Quantifiers Informally

```
Theorem: ∀x:t.P(x)

Proof:
Consider arbitrary x of type t,
  proof of P(x)
  ...
```

Universal Quantifiers in Lean:

```
Theorem foo : ∀x:t.P(x) := by

                        |  |- ∀x:t.P(x)

  intro x               |   x:t |- P(x)
  proof of P(x)
  ...
```

Intro rule for universal quantifiers converts one goal ∀x:t.P(x) into two things:

- an assumption we can use x : t
- a new goal P(x) we must prove

Proof Tactics: Implication

Proofs of Implications
On paper

```
Theorem: P -> Q

Proof:
assume P (call this assumption h)
prove Q:
  ...
  P      (by h!)
```

Proofs of Implications
In Lean:

```
Theorem foo : P -> Q := by

                    |    |- P -> Q

intro h             |    h:P |- Q

...                 |    h:P |- P
exact h
```

Intro rule for implication converts a single goal P -> Q into two things:

- an assumption P we can use
- a new goal Q we must prove

Proof Tactics: Implication

Proofs of Implications
On paper

```
Theorem: P -> Q

Proof:
assume P (call this assumption h)
prove Q:
  ...
  P      (by h!)
```

Proofs of Implications
In Lean:

```
Theorem foo : P -> Q := by

                    |    |- P -> Q

intro h             |    h:P |- Q

...                 |    h:P |- P
exact h
```

The intro rules for P -> Q and ∀x:t.P(x) are suspiciously similar! Both:

- introduce a hypothesis
- generate a goal

You would not be wrong to think they are connected!

Proof Tactics: Cases

Proof by cases
On paper

```
Theorem: ∀x:bool.P(x)

Proof:
Consider arbitrary x:bool

Proof by cases on x:

case x = true:
  proof of P(true)

case x = false
  proof of P(false)

done
```

Proof by cases
In Lean:

```
Theorem foo : ∀x:bool.P(x) := by

                                | |- ∀x:t.P(x)

intro x                         | x:t |- P(x)

cases x with

| true =>                       | |- P(true)
   ... proof of P(true) ...

| false =>                      | |- P(false)
   ... proof of P(false) ...
```

Notice that in each case, the hypothesis x disappears and we substitute the value v for that case for x.

We have a different set of cases depending on the type of each value

MORE PROOFS

A Surprising Situation

```
inductive Nat : Type where
  | zero
  | succ (n : Nat)

def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

for all natural numbers x,
add x zero = x

Easy.

for all natural numbers x,
add zero x = x

Harder.

The Easy Proof

```
inductive Nat : Type where
  | zero
  | succ (n : Nat)

def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

Theorem: for all natural numbers x, add x zero = x

Proof:
Consider arbitrary natural number x,
add x zero (LHS)
= match zero with | zero => x | succ m' => ... (eval add)
= x (eval add)
QED! (observe that LHS = RHS!)

The Harder Proof -- Attempt 1

```
inductive Nat : Type where
  | zero
  | succ (n : Nat)

def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

Theorem: for all natural numbers x, add zero x = x

Proof:
Consider arbitrary x,
add zero x (LHS)
= match x with | zero => zero | succ m' => ... (eval add)
= ??? (how to evaluate when our argument is an arbitrary value x? We can't represent "the" answer.)

The Harder Proof -- Attempt 2

```
inductive Nat : Type where
  | zero
  | succ (n : Nat)

def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

Theorem: for all natural numbers x, add zero x = x

Proof:
Consider arbitrary x. Proof by cases.
case x = zero:

add zero zero	**(RHS)**
= match zero with | zero => zero | ...	**(eval)**
= zero	**(eval)**

case done!

The Harder Proof -- Attempt 2 Continued

```
inductive Nat : Type where
  | zero
  | succ (n : Nat)

def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

Theorem: for all natural numbers x, add zero x = x

Proof:
Consider arbitrary x. Proof by cases. case x = zero: ...

case x = succ y:

```
  add zero (succ y)                                                    (RHS)
= match succ y with | zero => zero | succ m' => succ (add zero m')    (ev)
= succ (add zero y)                                                    (ev)
= succ (match y with | ...)                                            (ev)
= ??? stuck again ?
```

The Harder Proof -- Attempt 3?

```
inductive Nat : Type where
  | zero
  | succ (n : Nat)

def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

Theorem: for all natural numbers x, add zero x = x

Proof:
Consider arbitrary x
case x = zero: ...
case x = succ y
subcase y = zero
subcase y = succ z
subsubcase z = zero
subsubcase z = succ w
subsubsubcase ...

there are infinitely many cases of this kind. we need a way to cover all such cases!

Induction: A more sophisticated "cases"

When proving theorems about recursive functions, we usually need **induction**.

Proofs of property P(n) ***by induction over the natural numbers n***, have two cases:

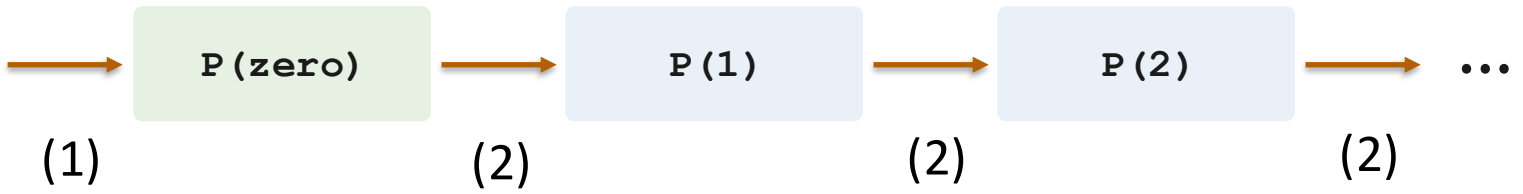
(1) case n = zero

- **must prove P(zero)**

(2) case n = succ k, for an arbitrary natural number k.

- **assume P(k)**
- **then prove P(succ k)**

If someone requests evidence P(i) holds for a specific natural number i, we can supply that evidence. We use the proof in case (1) to supply the initial evidence, and the proof in case (2) to supply evidence for each additional step in the chain:



Back to the proof

Theorem: For all natural numbers n,
`add zero n = n`

```
def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

Proof: **By induction on n.**

Case n = zero:

```
   add zero zero                    (LHS)
= zero                              (evaluation of add)
```

Back to the proof

Theorem: For all natural numbers n,
`add zero n = n`

```
def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

n + m is notation for add n m

Proof: **By induction on n.**

Case n = zero:

```
    add zero zero
=   zero                                (evaluation of add)
```

Case n = succ k: **IH: `add zero k = k`**

Back to the proof

Theorem: For all natural numbers n,
`add zero n = n`

```
def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

n + m is notation for add n m

Proof: **By induction on n.**

Case n = zero:

```
    add zero zero
=   zero                          (evaluation of add)
```

Case n = succ k: **IH: add `zero k = k`**

```
    add zero (succ k)             (LHS)
=   succ (add zero k)             (evaluation of add)
```

Back to the proof

Theorem: For all natural numbers n,
`add zero n = n`

```
def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

n + m is notation for add n m

Proof: **By induction on n.**

Case n = zero:

```
    add zero zero
=   zero                          (evaluation of add)
```

Case n = succ k: **IH: add `zero k = k`**

```
    add zero (succ k)
=   succ (add zero k)             (evaluation of add)
=   succ k                        (by IH! (and subst of equals for equals))
```

Back to the proof

Theorem: For all natural numbers n,
`add zero n = n`

```
def add (n : Nat) (m : Nat) : Nat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')
```

n + m is notation for add n m

Proof: **By induction on n.**

Case n = zero:

```
    add zero zero
=   zero                          (evaluation of add)
```

Case n = succ k: **IH: add `zero k = k`**

```
    add zero (succ k)
=   succ (add zero k)             (evaluation of add)
=   succ k                        (by IH!)
```

QED!

Template: Induction on natural numbers

Theorem: For all natural numbers n, P(n).

Proof: By induction on natural numbers n.

Case n = zero:

must prove: P(zero)

...

Case n = succ k: **IH: P**(k)

must prove: P(succ k)

...

INDUCTIVE PROOFS IN LEAN

Paper proofs and Lean proofs

On paper	In Lean
By induction on n.	`induction n with`
Case n = zero:	`| zero =>`
Case n = succ k: IH: ...(k)...	`| succ k ih =>`
(by def of f)	`rfl`, or `rewrite` with an equation lemma such as `add_succ`
(by IH)	`rewrite [ih]` or `exact ih`
case done!	the goal closes (`rfl`, `exact`)
cases must cover all natural numbers	Lean checks: one branch per constructor

Summary

Proofs about programs are structured like the programs:

types tell you the kinds of values your proofs operate over,

types suggest how to break proofs into cases,

recursion on smaller values means appealing to the IH on smaller values.

Key proof ideas:

substitute equals for equals;

calculate (evaluate) to prove simple equalities;

use induction to prove properties of recursive functions.

In Lean:

`induction` makes one subgoal per constructor, with an IH for the smaller value;

`rewrite`, `rfl` and `exact` are used in the body of the proof