PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 5 of 10

More on Proofs

"The only way to learn mathematics is to do mathematics."

— Paul Halmos

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/

Section 1

Introduction

Last Week: Rida's Session

What Rida covered in Week 4:

- Basic tactics: `rfl`, `exact`, `intro`, `apply`
- Rewriting with `rw` and `symm`
- Function extensionality (`ext`)
- Structuring proofs with `refine`
- Destructuring with `obtain`
- Mathematical induction on natural numbers
- Calculational proofs with `calc`
- List permutations and orderings

Great foundation! Now we build on it. Today we'll see how these tactics fit into the larger idea of mathematical proof, and learn to navigate Lean's massive library of existing theorems.

This Week: Becoming a Theorem Prover

Today's Goal: Prepare you to prove real theorems

What we'll cover:

- Review of basic tactics with new examples
- Finding and using library theorems
- Logical connectives more in depth
- Structured proofs with `have` and `show`
- List proofs (new examples!)
- Classical reasoning
- Writing clean, readable proofs

Both styles can be mixed and are equally powerful!

Tactic Mode Syntax

Keyword: by indicates tactical proof

```
theorem example_theorem :
    statement := by
  tactic1
  tactic2
  ...
  tacticN
  done
```

Each tactic transforms the proof state – the collection of hypotheses and goals. Tactics are commands that tell Lean how to manipulate the proof state to construct the final proof term.

Next week: Rida will introduce you to contributing to Mathlib!

Section 2

Review: Basic Tactics

Quick Review of Last Week's Tactics

Core tactics Rida introduced:

Proving equality:

- `rfl` – reflexivity
- `exact` – provide proof term
- `rw` – rewriting
- `symm` – symmetry

Structural tactics:

- `intro` – introduce hypothesis
- `apply` – backward reasoning
- `refine` – partial proof with holes

Pattern matching:

- `obtain` – destructure $\exists$
- `ext` – extensionality
- `cases` – case analysis
- `induction` – induction

Calculational:

- `calc` – chains of equalities

Today: We'll see these in action with more complex examples and learn when to use each one!

Review: `rfl` and `exact`

`rfl`: Proves definitional equality (things equal by computation)

```
example (n : ℕ) : n = n := by rfl

example : 2 + 2 = 4 := by rfl  -- computation!

example (f : ℕ → ℕ) (x : ℕ) : (fun y => f y) x = f x := by
  rfl  -- β-reduction
```

`exact`: Provide exact proof term

```
example (P Q : Prop) (h : P ∧ Q) : Q ∧ P := by
  exact And.comm.mp h

example (n m : ℕ) (h : n = m) : m = n := by
  exact h.symm
```

Key difference: `rfl` proves things by computation, `exact` uses an existing proof. Both close the goal immediately – no subgoals remain.

More `rfl` Examples

Question: What counts as definitional equality?

```
-- Unfolding definitions
example (n : ℕ) : n + 0 = n := by rfl

-- List operations
example : [1, 2] ++ [3] = [1, 2, 3] := by rfl

-- Function composition
example (f g : ℕ → ℕ) (x : ℕ) : (f ∘ g) x = f (g x) := by rfl

-- Pattern matching
example (n : ℕ) : (match n with | 0 => 5 | _ => 10) =
                  (if n = 0 then 5 else 10) := by
  cases n <;> rfl
```

Remember: If Lean can compute both sides to the same value, `rfl` works! This includes unfolding definitions, β-reduction, and pattern matching.

Review: `intro` and `apply`

`intro`: Move hypothesis from goal to context

```
example (P Q : Prop) : P → Q → P := by
  intro hP
  intro hQ
  exact hP

-- Can also intro multiple at once
example (P Q R : Prop) : P → Q → R → P := by
  intro hP hQ hR
  exact hP
```

`apply`: Backward reasoning (match goal with conclusion)

```
example (a b c : ℕ) (h1 : a < b) (h2 : b < c) : a < c := by
  apply Nat.lt_trans
  · exact h1  -- prove first premise: a < b
  · exact h2  -- prove second premise: b < c
```

Pattern: `intro` for $\rightarrow$ and $\forall$ in goal. `apply` when goal matches conclusion of a theorem.

More `intro` and `apply` Examples

`intro` with pattern matching:

```
example (P Q : Prop) : P ∧ Q → Q ∧ P := by
  intro (hp, hq)  -- destructure immediately!
  exact (hq, hp)

example : ∀ (n : ℕ), n < n + 1 := by
  intro n
  exact Nat.lt_succ_self n
```

`apply` with partial instantiation:

```
example (a b c d : ℕ) (h1 : a ≤ b) (h2 : c ≤ d) : a + c ≤ b + d := by
  apply Nat.add_le_add
  · exact h1
  · exact h2

-- apply can also leave holes
example (n : ℕ) : n + 1 > n := by
  exact Nat.lt_succ_self n
```

Review: `rw` (Rewriting)

Purpose: Rewrite using equalities

```
example (a b c : ℕ) (h1 : a = b) (h2 : b = c) : a + 1 = c + 1 := by
  rw [h1]      -- rewrite a to b: goal is now b + 1 = c + 1
  rw [h2]      -- rewrite b to c: goal is now c + 1 = c + 1
  rfl          -- now trivial
```

Advanced features:

```
example (a b : ℕ) (h : b = a + 1) : a + 1 = b := by
  rw [← h]     -- backward rewrite (arrow points left!)

example (a b c : ℕ) (h : a = b) : a + c = b + c := by
  rw [h] at *  -- rewrite everywhere

example (xs : List ℕ) (h : xs.length = 5) : xs.length + 1 = 6 := by
  rw [h]       -- can rewrite in complex expressions
  rfl
```

Remember: Can rewrite in hypotheses too with `rw [h] at hyp`!

More `rw` Examples

Chaining rewrites:

```
example (a b c d : ℕ) (h1 : a = b) (h2 : b = c) (h3 : c = d) : a = d := by
  rw [h1, h2, h3]  -- chain multiple rewrites!

-- With library theorems
example (n m : ℕ) : n + m = m + n := by
  rw [Nat.add_comm]
```

Selective rewriting:

```
example (n : ℕ) : n + n = 2 * n := by
  rw [two_mul]  -- only rewrites 2 * n, not n + n

-- Rewrite at specific position
example (a b : ℕ) (h : a = b) : a + a = b + a := by
  nth_rewrite 1 [h]  -- rewrites first occurrence
  rfl
  -- goal: b + a = b + a
```

Important! `rw` is one of the most commonly used tactics – master it!

Review: `induction` and `calc`

`induction`: Prove by mathematical induction

```
theorem add_zero (n : ℕ) : n + 0 = n := by
  induction n with
  | zero => rfl  -- base case
  | succ n ih =>
      -- inductive hypothesis: ih : n + 0 = n
      -- goal: succ n + 0 = succ n
      rw [Nat.add_succ, ih]
```

`calc`: Chain equalities/inequalities

```
example (a b c : ℕ) (h1 : a = b) (h2 : b < c) : a < c := calc
  a = b := h1
  _ < c := h2
```

Both are essential and are often used together: induction provides structure, calc provides syntactic clarity.

More `induction` Examples

Performing induction on lists:

```
theorem length_append' {α : Type*} (xs ys : List α) :
    (xs ++ ys).length = xs.length + ys.length := by
  induction xs with
  | nil => simp
  | cons x xs ih =>
      simp only [List.cons_append, List.length_cons]
      rw [ih]
      omega
```

Performing induction with multiple variables:

```
theorem my_add_comm (n m : ℕ) : n + m = m + n := by
  induction m with
  | zero => omega
  | succ m ih =>
      rw [Nat.add_succ, ih, Nat.succ_add]
```

Section 3

Finding Library Theorems

The Library is Vast

Issue: Lean's mathematical library (Mathlib) has over 100,000 theorems

The bad news (boo):

- You can't memorize them all
- You won't know what exists at first

The good news (yay):

- Systematic naming conventions make theorems discoverable
- Powerful search tools automate the hunting process
- Lean can help you find what you need!

Now: You have to be a theorem hunter. Most of your time will actually be spent *finding* theorems rather than proving from scratch – this is normal and expected in real mathematical work!

Naming Conventions

Naming Pattern (Convention): `conclusion_of_premise_of_premise`

Examples:

```
#check List.length_append  -- length (xs ++ ys) = length xs + length ys
#check List.reverse_reverse -- reverse (reverse xs) = xs
#check List.map_append      -- map f (xs ++ ys) = map f xs ++ map f ys
#check List.append_nil      -- xs ++ [] = xs
#check List.nil_append      -- [] ++ xs = xs
```

Out loud, they read as:

- "length append": length of append is...
- "reverse reverse": reverse of reverse is...
- "append nil": append with nil gives...

The pattern is consistent across all of Mathlib: operations come first, then properties or results. Once you internalize this, finding theorems becomes intuitive.

More Naming Convention Examples

Some arithmetic operations in Lean:

```
#check Nat.add_comm        -- n + m = m + n
#check Nat.add_assoc       -- (n + m) + k = n + (m + k)
#check Nat.mul_comm        -- n * m = m * n
#check Nat.add_zero        -- n + 0 = n
#check Nat.zero_add        -- 0 + n = n
```

Some relations and orderings in Lean:

```
#check Nat.le_trans        -- a ≤ b → b ≤ c → a ≤ c
#check Nat.lt_of_le_of_lt  -- a ≤ b → b < c → a < c
#check Nat.add_le_add      -- a ≤ b → c ≤ d → a + c ≤ b + d
```

Notice the Pattern: It's always [operation name], then [what it does] or [what's true about it]. For two premises, you connect them with [`_of_`].

Using Ctrl+Space

Type partial name and hit `Ctrl+Space` (in VSCode):

```
example (xs ys : List ℕ) : (xs ++ ys).length = xs.length + ys.length := by
  -- Type "List.length_" and hit Ctrl+Space
  -- See: length_append, length_cons, length_map, etc.
  exact List.length_append xs ys
```

For reverse:

```
example (xs : List ℕ) : xs.reverse.reverse = xs := by
  -- Type "List.reverse_" and hit Ctrl+Space
  exact List.reverse_reverse xs
```

Pro tip: Always try typing what you think the theorem should be called! The autocomplete will show you all matching theorems. This is faster than searching documentation and helps you learn and internalize the Lean theorem naming patterns.

The `exact?` Tactic

Let Lean search matching theorems for you:

```
example (xs ys : List ℕ) : xs ++ [] = xs := by
  exact?  -- Suggests: exact List.append_nil xs

example (xs : List ℕ) : [] ++ xs = xs := by
  exact?  -- Suggests: exact List.nil_append xs

example (xs ys zs : List ℕ) :
    (xs ++ ys) ++ zs = xs ++ (ys ++ zs) := by
  exact?  -- Suggests: exact List.append_assoc xs ys zs
```

Not always, but the times you should use `exact?` are:

- When you are confident the theorem already exists
- For learning naming patterns
- During exploration

REMEMBER: Replace `exact?` with the actual theorem in final code! The `?` tactics are for development only – they search through the library at compile time.

More "Search" Tactics

`apply?`: Extremely similar to `exact?`, but used for backward reasoning

```
example (a b c : ℕ) (h1 : a < b) (h2 : b < c) : a < c := by
  apply?  -- Suggests: apply Nat.lt_trans
  · exact h1
  · exact h2
```

`rw?`: Suggests possible rewrites

```
example (n m : ℕ) : n + m + 0 = m + n := by
  rw?  -- Suggests various rewrites including add_zero, add_comm
```

These are your research assistants! Use them freely during development, then replace with concrete tactics for the final proof.

Exploring with #check

Investigate types and theorems:

```
#check List.map
-- List.map : {α β : Type*} → (α → β) → List α → List β

#check List.length_map
-- List.length_map : ∀ {α β : Type*} (f : α → β) (l : List α),
--   (l.map f).length = l.length

#check List.map_map
-- List.map_map : ∀ {α β γ : Type*} (g : β → γ) (f : α → β) (l : List α),
--   map g (map f l) = map (g ∘ f) l
```

Pattern: Use #check before proving to see what's available! The type signature tells you exactly how to apply the theorem. The "" indicate implicit arguments that Lean will infer automatically.

Section 4

Logical Connectives

Logic: The Language of Mathematics

Everything in mathematics is built from logical connectives

The fundamental building blocks:

- $\forall$ (for all) – universal quantification
- $\rightarrow$ (implies) – implication
- $\wedge$ (and) – conjunction
- $\vee$ (or) – disjunction
- $\exists$ (exists) – existential quantification
- $\leftrightarrow$ (iff) – bi-implication
- $\neg$ (not) – negation

Each has:

- **Introduction rules**: how to prove it
- **Elimination rules**: how to use it

Conjunction ($\wedge$): And (1)

To prove a statement like $p \wedge q$, we must independently provide a proof for p and a proof for q.

Introduction: In tactic mode, the `constructor` tactic recognizes this and creates two separate subgoals. Alternatively, the anonymous constructor `⟨ ·, · ⟩` lets us bundle the two proofs concisely in term mode.

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
  constructor
  · exact hp
  · exact hq

-- More concisely with anonymous constructor:
example (p q : Prop) (hp : p) (hq : q) : p ∧ q :=
  ⟨hp, hq⟩
```

Conjunction ($\wedge$): And (2)

What if we already know $p \wedge q$ is true? We can extract its individual components to help prove new goals.

Elimination: If we have a hypothesis `h : p ∧ q`, we access the left side using `h.left` (or `h.1`) and the right side using `h.right` (or `h.2`). Let's use this to swap the order of a conjunction.

```
example (p q : Prop) (h : p ∧ q) : q ∧ p := by
  constructor
  · exact h.right  -- or h.2
  · exact h.left   -- or h.1
```

More Conjunction Examples (1)

Nested conjunctions:

```
example (p q r : Prop) (hp : p) (hq : q) (hr : r) : (p ∧ q) ∧ r := by
  constructor
  · constructor
    · exact hp
    · exact hq
  · exact hr

-- Or more concisely:
example (p q r : Prop) (hp : p) (hq : q) (hr : r) : (p ∧ q) ∧ r :=
  ((hp, hq), hr)
```

Extracting from nested conjunctions:

```
example (p q r : Prop) (h : (p ∧ q) ∧ r) : q := by
  exact h.left.right  -- navigate the structure!

example (p q r : Prop) (h : (p ∧ q) ∧ r) : q :=
  h.1.2  -- using numerical accessors
```

Disjunction ($\vee$): Or (1)

Unlike conjunction, proving a disjunction like $p \vee q$ only requires us to prove *one* of the components.

Introduction: We use the `left` tactic to tell Lean we want to prove p, or the `right` tactic to tell Lean we want to prove q.

```
example (p : Prop) (hp : p) : p ∨ q := by
  left
  exact hp

example (q : Prop) (hq : q) : p ∨ q := by
  right
  exact hq
```

Disjunction ($\vee$): Or (2)

What if we are given a disjunction as a hypothesis? Since we don't know which side is actually true, we must prove our goal works for *both* possibilities.

Elimination: We use the `cases` tactic to split the proof into two distinct branches (as Rida showed!).

```
example (p q r : Prop) (h : p ∨ q) (hp : p → r) (hq : q → r) : r := by
  cases h with
  | inl hp' => exact hp hp'
  | inr hq' => exact hq hq'
```

More Disjunction Examples

Nested disjunctions:

```
example (p q r : Prop) (h : (p ∨ q) ∨ r) : r ∨ (q ∨ p) := by
  cases h with
  | inl hpq =>
      cases hpq with
      | inl hp => right; right; exact hp
      | inr hq => right; left; exact hq
  | inr hr => left; exact hr
```

Combining with other connectives:

```
example (p q : Prop) : p ∧ (q ∨ r) → (p ∧ q) ∨ (p ∧ r) := by
  intro ⟨hp, hqr⟩
  cases hqr with
  | inl hq => left; exact ⟨hp, hq⟩
  | inr hr => right; exact ⟨hp, hr⟩
```

This is the distributivity of $\wedge$ over $\vee$!

Biconditional ($\leftrightarrow$): If and Only If (1)

To prove a biconditional $p \leftrightarrow q$, we must show that the two statements are logically equivalent. This requires proving two separate implications: $p \to q$ and $q \to p$.

Introduction: Just like with conjunction ($\wedge$), we use the `constructor` tactic to split the goal into two pieces: the forward and backward directions.

```
example (p q : Prop) : p ∧ q ↔ q ∧ p := by
  constructor
  · intro h
    exact (h.right, h.left)
  · intro h
    exact (h.right, h.left)
```

Biconditional ($\leftrightarrow$): If and Only If (2)

If we are given an equivalence as a hypothesis, we can use it as a two-way street to convert a proof of one side into a proof of the other.

Elimination: We extract the forward direction using `.mp` (modus ponens) and the backward direction using `.mpr` (modus ponens reversed).

```
example (p q : Prop) (h : p ↔ q) (hp : p) : q :=
  h.mp hp  -- modus ponens

example (p q : Prop) (h : p ↔ q) (hq : q) : p :=
  h.mpr hq  -- modus ponens reversed
```

More Biconditional Examples

Chaining biconditionals:

```
example (p q r : Prop) (h1 : p ↔ q) (h2 : q ↔ r) : p ↔ r := by
  constructor
  · intro hp
    exact h2.mp (h1.mp hp)
  · intro hr
    exact h1.mpr (h2.mpr hr)
```

Using biconditionals for rewriting:

```
example (p q : Prop) (h : p ↔ q) : (p ∧ r) ↔ (q ∧ r) := by
  constructor
  · intro ⟨hp, hr⟩
    exact ⟨h.mp hp, hr⟩
  · intro ⟨hq, hr⟩
    exact ⟨h.mpr hq, hr⟩

-- Can also use rw with biconditionals!
example (p q : Prop) (h : p ↔ q) : (p ∧ r) ↔ (q ∧ r) := by
  rw [h]
```

Negation ($\neg$): Not

In Lean, negation is not a primitive concept (derived concept). The statement $\neg p$ is strictly defined as an implication: $p \to \bot$ (`False`).

Introduction: To prove $\neg p$, we assume p is true (using the `intro` tactic) and show that this assumption leads to a contradiction. Let's prove the Law of Noncontradiction:

```
example (p : Prop) : ¬(p ∧ ¬p) := by
  intro h
  -- h : p ∧ ¬p
  -- Goal: False
  have hp : p := h.left
  have hnp : ¬p := h.right
  exact hnp hp  -- Apply ¬p to p to get False
```

Negation ($\neg$): Not (2)

Because $\neg p$ is structurally just $p \to \bot$, we can work with negated hypotheses exactly like we work with functions or implications.

Working with Negations: If our goal is a negation, we start with `intro`. If we have a negated hypothesis, we apply it to a proof of its positive counterpart.

```
example (p q : Prop) (h : ¬q) (hpq : p → q) : ¬p := by
    intro hp
    have hq : q := hpq hp
    exact h hq
```

More Negation Examples

Double negation introduction:

```
example (p : Prop) (hp : p) : ¬¬p := by
  intro hnp
  exact hnp hp

-- This works constructively!
```

De Morgan's Laws (one direction):

```
example (p q : Prop) : ¬(p ∧ q) → ¬p ∨ ¬q := by
  intro h
  by_cases hp : p
  · right
    intro hq
    exact h (hp, hq)
  · left
    exact hp

-- Note: This needs classical logic (by_cases)!
```

Existential (∃): There Exists (1)

To prove a statement like $\exists x, P(x)$, we must provide a specific example (called a "witness") that makes the property $P(x)$ true.

Introduction: We use the `use` tactic to supply this witness. Lean will then ask us to prove that our chosen witness actually works.

```
example : ∃ x : ℕ, x + 5 = 8 := by
  use 3
  rfl

example (xs : List ℕ) (h : xs ≠ []) :
    ∃ y ys, xs = y :: ys := by
  cases xs with
  | nil => contradiction
  | cons y ys => use y, ys
```

Existential (∃): There Exists (2)

If we have an existential statement as a hypothesis, we know a witness exists, but we don't know its exact value. We must unpack it to use it in our proof.

Elimination: As Rida showed, we use the `obtain` tactic to extract the "mystery" witness and assign it a name, alongside the proof of its property.

```
example (h : ∃ x : ℕ, x > 5) : ∃ x : ℕ, x > 3 := by
  obtain ⟨x, hx⟩ := h
  use x
  omega
```

More Existential Examples

Multiple existentials:

```
example : ∃ x y : ℕ, x + y = 10 ∧ x * y = 21 := by
  use 3, 7
  constructor
  · rfl
  · rfl
```

Existential with computation:

```
example (n : ℕ) : ∃ m : ℕ, m = n + 1 := by
  use n + 1

example (xs ys : List ℕ) : ∃ zs : List ℕ, zs.length = xs.length + ys.length := by
  use xs ++ ys
  exact List.length_append
```

The witness can be an arbitrary expression – Lean will compute it!

Section 5

Structured Proofs

Writing Readable Proofs

Good proofs are like good code:

- Clear structure
- Named intermediate results
- Explicit reasoning steps
- Easy to follow and modify

Two key tactics for structure:

- `have`: State and prove intermediate claims
- `show`: Make current goal explicit

Why this matters: When you submit proofs, people need to read them! Code review for mathematical proofs is just as important as for software. Reviewers will thank you for clear, well-structured proofs with named intermediate steps.

The `have` Tactic

Purpose: Introduce intermediate facts

```
example (xs ys : List ℕ) (h : xs ++ ys = []) : xs = [] ∧ ys = [] := by
  have hxs : xs = [] := by
    cases xs with
    | nil => rfl
    | cons x xs' =>
      simp only [List.cons_append] at h
      contradiction
  have hys : ys = [] := by
    cases ys with
    | nil => rfl
    | cons y ys' =>
      rw [hxs] at h
      simp only [List.nil_append] at h
      contradiction
  exact ⟨hxs, hys⟩
```

Better than: Nested tactics with no names. Each `have` gives a name to a fact you'll use later. This makes proofs self-documenting and easier to debug when something goes wrong.

Complex `have` Example

Building up a calculation (notice how intermediate steps are explicitly named and proven):

```
example (xs ys : List ℕ) :
    (xs ++ ys).reverse = ys.reverse ++ xs.reverse := by
  induction xs with
  | nil =>
      have h1 : ([] ++ ys).reverse = ys.reverse := by
        simp only [List.nil_append]
      have h2 : ys.reverse ++ [].reverse = ys.reverse := by
        simp only [List.reverse_nil, List.append_nil]
      rw [h1, h2]
  | cons x xs' ih =>
      have h1 : ((x :: xs') ++ ys).reverse
                = (x :: (xs' ++ ys)).reverse := by
        simp only [List.cons_append]
      have h2 : (x :: (xs' ++ ys)).reverse
                = (xs' ++ ys).reverse ++ [x] := by
        simp only [List.reverse_cons]
      have h3 : (xs' ++ ys).reverse
                = ys.reverse ++ xs'.reverse := ih
      rw [h1, h2, h3, List.append_assoc]
      simp only [List.reverse_cons]
```

This makes it clear what transformations we're applying and in what order.

More **have** Examples

Using **have** for case splitting:

```
example (n : ℕ) : n = 0 ∨ n > 0 := by
  have h : n = 0 ∨ n ≠ 0 := em (n = 0)
  cases h with
  | inl h => left; exact h
  | inr h =>
      right
      omega
```

have without proof (letting Lean infer):

```
example (n m : ℕ) (h : n + m = 10) : m + n = 10 := by
  have : n + m = m + n := Nat.add_comm n m
  rw [← this]
  exact h

-- Lean can often infer the type of have
```

The `show` Tactic

Purpose: Make the goal explicit

```
example (xs : List ℕ) : xs.reverse.reverse = xs := by
  induction xs with
  | nil =>
      show [].reverse.reverse = []
      rfl
  | cons x xs' ih =>
      show (x :: xs').reverse.reverse = x :: xs'
      simp only [List.reverse_cons, List.reverse_append]
      rw [ih]
      rfl
```

When to use: When goal is complex or changes. The `show` tactic doesn't change anything, it just makes the goal explicit. This is helpful in induction where the goal changes subtly between cases, and readers need to see exactly what you're proving.

Combining have and show

Using both correctly, makes the proof's structure very clear:

```
example (xs ys : List ℕ) (h : xs.length = ys.length) :
example (xs ys : List ℕ) (h : xs.length = ys.length) :
    (xs ++ ys).length = 2 * xs.length := by
  have hlen : (xs ++ ys).length = xs.length + ys.length :=
    List.length_append
  rw [hlen]
  show xs.length + ys.length = 2 * xs.length
  rw [h]
  omega
```

Benefits:

- Each step is named
- Goal is made explicit
- Easy to understand and debug

This proof reads like a mathematical argument: "First, observe that... Now we need to show... Substituting and simplifying gives the result."

Section 6

List Proofs

Why Lists?

Lists are fundamental in mathematics and functional programming

Why practice with lists:

- Simple inductive structure
- Many many useful theorems
- Models: vectors, paths (graphs), sequences, strings, etc.
- Extremely common in real formalization work (e.g. Mathlib)

Today's examples:

- List append properties
- List reverse properties
- List map properties
- List length relationships

Note: Lists are the perfect training ground for inductive proofs. Once you master say list proofs, you can tackle more complex structures like trees, graphs, and program syntax.

List Append is Associative (Theorem)

Theorem: `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`

```
theorem my_append_assoc (xs ys zs : List α) :
    (xs ++ ys) ++ zs = xs ++ (ys ++ zs) := by
  induction xs with
  | nil =>
      show ([] ++ ys) ++ zs = [] ++ (ys ++ zs)
      rfl
  | cons x xs' ih =>
      show ((x :: xs') ++ ys) ++ zs
           = (x :: xs') ++ (ys ++ zs)
      simp only [List.cons_append]
      rw [ih]
```

Pattern: Induction on first list, `simp` handles computation

Note: This is actually `List.append_assoc` in Mathlib! Associativity is crucial for rewriting – it means we can rearrange parentheses freely in chains of appends. This is why we can write `xs ++ ys ++ zs` without ambiguity.

List Map Preserves Length (Theorem)

Theorem: Mapping doesn't change length (i.e. `map` is a "length-preserving" operation)

```
theorem my_length_map {α β : Type*} (f : α → β) (xs : List α) :
    (xs.map f).length = xs.length := by
  induction xs with
  | nil => rfl
  | cons x xs' ih =>
      simp only [List.map_cons, List.length_cons]
      rw [ih]
```

Key tactics:

- Induction on the list
- Use library theorem `List.map_cons`
- `simp` with induction hypothesis

More List Examples: Filter

Theorem: Filter preserves or reduces length

```
theorem my_length_filter_le {α : Type*} (p : α → Bool) (xs : List α) :
    (xs.filter p).length ≤ xs.length := by
  induction xs with
  | nil => simp
  | cons x xs ih =>
      by_cases h : p x
      · simp only [List.filter_cons_of_pos h, List.length_cons]
        exact Nat.succ_le_succ ih
      · simp only [List.filter_cons_of_neg h, List.length_cons]
        exact Nat.le_succ_of_le ih
```

Pattern: Induction with case analysis on the predicate. When p x is true, the element stays (length increases by 1 on both sides). When false, it's filtered out (only original length increases).

Map Fusion (Theorem)

Theorem: Composing maps can be fused

```
theorem my_map_map {α β γ : Type*} (g : β → γ) (f : α → β) (xs : List α) :
    (xs.map f).map g = xs.map (g ∘ f) := by
  induction xs with
  | nil => rfl
  | cons x xs' ih =>
      simp only [List.map_cons]
      rw [ih]
      rfl
```

This is an optimization! One pass instead of two. Map fusion is a classic FP transformation: instead of traversing the list twice (i.e. once for `f`, once for `g`), we traverse once with the composition. Compilers use this theorem to optimize code!

Reverse and Append (Theorem)

Theorem: Reverse distributes over append

```
theorem my_reverse_append {α : Type*} (xs ys : List α) :
    (xs ++ ys).reverse = ys.reverse ++ xs.reverse := by
  induction xs with
  | nil =>
      show ([] ++ ys).reverse = ys.reverse ++ [].reverse
      simp only [List.nil_append, List.reverse_nil, List.append_nil]
  | cons x xs' ih =>
      simp only [List.cons_append, List.reverse_cons]
      rw [ih, List.append_assoc]
```

This shows that reverse “flips” the order of an append. Notice how we carefully build up the proof step by step using have – each transformation is explicit and justified.

A Challenge: Filter Length

Try this yourself:

```
-- Filtering a list gives a shorter or equal length list
theorem my_length_filter {α : Type*} (p : α → Bool) (xs : List α) :
    (xs.filter p).length ≤ xs.length := by
  sorry

-- Hint: Induction on xs
-- Use: List.filter_cons_of_pos and List.filter_cons_of_neg
-- Use: Nat.succ_le_succ for the recursive case
```

Steps:

- Induction on `xs`
- Nil case: trivial
- Cons case: use `cases` on `p x`

Filtering only removes elements, never adds them, so the filtered list is always at most as long as the original. (Hint: you need to case on whether the predicate is true or false for the head element)

Section 7

Classical Reasoning

Classical vs. Constructive Logic

Two "flavors" of logic:

Constructive logic (default in Lean):

- To prove $\exists x, P(x)$, must provide witness
- To prove $P \vee Q$, must prove which one
- Very strong – proofs contain computational content

Classical logic (with additional axiom):

- Can use Law of Excluded Middle: $P \vee \neg P$
- Can use proof by contradiction freely
- Closer to traditional mathematics

Lean supports both! Use `Classical` namespace when needed. The distinction matters in theoretical computer science – constructive proofs can be extracted to executable programs, while classical proofs cannot. Most everyday mathematics uses classical logic.

Law of Excluded Middle

The classical axiom:

```
open Classical

example (p : Prop) : p ∨ ¬p :=
  em p  -- law of excluded middle

-- Using it for case analysis
example (xs : List ℕ) : xs = [] ∨ ∃ y ys, xs = y :: ys := by
  cases em (xs = []) with
  | inl h => left; exact h
  | inr h =>
      right
      cases xs with
      | nil => contradiction
      | cons y ys => use y, ys
```

Note: Rida used `by_cases` which is the tactic version! The law of excluded middle says every proposition is either true or false – no middle ground. This seems obvious but is actually a strong assumption that constructive mathematics rejects.

Proof by Contradiction

Pattern: Assume negation, derive contradiction

```
open Classical

example (p q : Prop) (h : ¬p → q) (hnq : ¬q) : p := by
  by_contra hnp
  -- hnp : ¬p
  -- Goal: False
  have hq : q := h hnp
  exact hnq hq
```

When to use:

- When direct proof is hard
- When dealing with negations
- In traditional mathematics arguments

Tip: Try direct proof first, use contradiction as fallback. Proof by contradiction is powerful but can obscure the mathematical insight. Direct proofs are often more informative, showing *why* something is true rather than just that it *must be* true.

More Classical Examples (1)

The standard contrapositive ($p \to q$ implies $\neg q \to \neg p$) is actually constructive! But the *reverse* requires classical logic.

Classical Contrapositive: $(\neg q \to \neg p) \to (p \to q)$

```
open Classical

example (p q : Prop) : (p → q) → (¬q → ¬p) := by
  intro hpq hnq hp
  exact hnq (hpq hp)

-- This works constructively! No classical logic needed.
```

More Classical Examples (2)

Pushing a negation inside an "And" strictly requires classical logic. If we know $p \wedge q$ is false, we don't necessarily have a computable way to know *which* one is false.

Classical De Morgan's Law: $\neg(p \wedge q) \to \neg p \vee \neg q$

```
example (p q : Prop) : ¬(p ∧ q) → ¬p ∨ ¬q := by
  intro h
  by_cases hp : p
  · right
    intro hq
    exact h ⟨hp, hq⟩
  · left
    exact hp

-- Reverse direction is constructive:
example (p q : Prop) : ¬p ∨ ¬q → ¬(p ∧ q) := by
  intro h ⟨hp, hq⟩
  cases h with
  | inl hnp => exact hnp hp
  | inr hnq => exact hnq hq
```

Double Negation Elimination

This is very classical BUT not constructive:

```
open Classical

-- This needs classical logic!
example (p : Prop) (h : ¬¬p) : p := by
  by_contra hnp
  exact h hnp

-- Or more directly:
example (p : Prop) (h : ¬¬p) : p :=
  Classical.byContradiction h
```

Why is this not constructive?

- From $\neg\neg P$ we can't construct evidence for P
- We only know "it's not the case that P is false"
- Classical logic treats this as equivalent to true (axiom)

Section 8

Writing Clean Proofs

Best Practices

What makes a good proof:

1. Structure

- Use `have` for intermediate steps
- Use `show` to make goals explicit
- One idea per line

2. Naming

- Descriptive hypothesis names
- `h1`, `h2` for short results
- `ih` for induction hypothesis

3. Comments

- Brief comments for non-obvious steps
- High-level strategy at start

4. Simplicity

- Use library theorems
- Don't reinvent the wheel

Proofs are written once but read many times. Invest in clarity now; save debugging time later.

Example: Bad vs. Good Proof (1)

Bad: Dense, unclear (says nothing about *how* it works)

```
theorem bad_proof (xs ys : List ℕ) :
    (xs ++ ys).length = xs.length + ys.length := by
  induction xs with | nil => simp | cons x xs' ih => simp only [List.cons_append, List.length_cons]; rw [ih];
  ↪  omega
```

Why though? Crammed formatting, high cognitive load, zero structural signposting, etc. How to do better?

- Taking the time to structure an induction block prevents future headache.
- Good proofs make the intermediate states obvious, even if the reader isn't actively running Lean in their editor.
- Use structured blocks, e.g. placing each branch of `with` on a new indented line, or using `case nil =>`, `case cons =>`), to visually separate base case from inductive step.

Example: Bad vs. Good Proof (2)

Good: Clear structure (makes each step explicit)

```
theorem good_proof (xs ys : List ℕ) :
    (xs ++ ys).length = xs.length + ys.length := by
  induction xs with
  | nil =>
      show ([] ++ ys).length = [].length + ys.length
      simp
  | cons x xs' ih =>
      show ((x :: xs') ++ ys).length
           = (x :: xs').length + ys.length
      simp only [List.cons_append, List.length_cons]
      rw [ih]
      omega
```

When the proof breaks (and it will when the library changes), you'll thank yourself for the clarity.

Real Example: A Well-Structured Proof

```
theorem my_reverse_involutive {α : Type*} (xs : List α) :
    xs.reverse.reverse = xs := by
  -- Strategy: induction on xs
  -- Base case: [] reverses to itself
  -- Inductive case: use distributivity of reverse over append
  induction xs with
  | nil =>
      show [].reverse.reverse = []
      rfl
  | cons x xs' ih =>
      -- Want: (x :: xs').reverse.reverse = x :: xs'
      simp only [List.reverse_cons, List.reverse_append]
      rw [ih]
      rfl
```

This proof starts with a comment explaining the strategy, uses `show` to make goals explicit, names each intermediate step with `have`, and concludes cleanly. Good stuff!

Common Proof Patterns

Common threads in Lean proofs:

Goal type	Common approach
`xs ++ ys = ys ++ xs`	Find associativity/commutativity theorems
`xs.reverse = ys`	Use reverse properties from library
`xs.length = n`	Induction or library theorem
`xs = []`	Cases on xs or contradiction
`x ∈ xs`	Use membership lemmas
`∀ x ∈ xs, P x`	Induction on xs

Note: Look for patterns, not just tactics! Experienced proof writers recognize these goal shapes instantly and know the standard approaches. Building this pattern recognition is more important than memorizing individual tactics.

Section 9

Summary

What We Covered

Major topics:

1. Tactic Review

- Basic tactics with new examples
- When to use each tactic

2. Finding Theorems

- Naming conventions
- `exact?` tactic and `#check`

3. Logic

- Logical connectives: $\wedge$, $\vee$, $\leftrightarrow$, $\neg$, $\exists$
- Introduction and elimination rules
- Classical reasoning

4. Structured Proofs

- `have` for intermediate steps
- `show` for explicit goals
- Clean, readable code

5. Lists

- Append, reverse, map properties
- Induction patterns

You now have the tools to read and write real mathematical proofs in Lean. These are the same techniques used in real verification projects and Mathlib contributions.

You're Ready!

Skills acquired:

- Navigate large libraries
- Understand logical structure
- Write clear, structured proofs
- Work with inductive data types
- Use classical reasoning when needed

Next steps:

- Practice these techniques
- Explore Mathlib documentation
- Try proving theorems on your own

Next week: Rida introduces Mathlib contributions! You'll learn how to find gaps in the library, write proofs that meet Mathlib standards, and submit your first pull request. The skills from today will be essential for that work.

Section 10

Assignments

This Week's Work

Readings

- Mathematics in Lean (Chapter 1 + at least one chapter covering a topic of your choosing, e.g. 2 and 3)
- Theorem Proving in Lean 4 (Chapters 7-8)
- The Hitchhiker's Guide to Logical Verification (Chapters 6-9)

Assignments

- The Natural Number Game
- Later: Programming Assignment 5 (List theorems)

Preparation for next week: Start browsing Mathlib on GitHub! Look at recent pull requests to see what kinds of contributions are being made. Understanding the library structure now will help you when you start contributing.

Questions & Discussion

Questions?

Join our community:
Discord: Link on website
WhatsApp: Link on website
Website: https://danieldia-dev.github.io/proofs/
Email: dmd13@mail.aub.edu

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 5 of 10

More on Proofs

"The only way to learn mathematics is to do mathematics."

— Paul Halmos

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/

Section 11

Optional: Operational Semantics

What is Formal Semantics?

Formal semantics: Mathematical specification of what programs mean

Why formalize?

- Specify programming languages precisely
- Reason about programs mathematically
- Build verified compilers, interpreters, analyzers
- Find bugs in language specifications

Success story: WebAssembly

- Formalization revealed multiple soundness bugs
- Fixed before becoming standards

Why Proof Assistants for Semantics?

Good match for automation:

- Little background machinery needed
- Lots of cases (computers excel at this)
- Track changes when extending language
- Catch subtle bugs

Real world: 30%+ of papers at the Principles of Programming Languages (POPL) conference use proof assistants

In this lecture: Build operational semantics for a simple language

The WHILE Language

Minimalistic imperative language:

$$
\begin{aligned}
S ::= \ & \texttt{skip} && \text{(no-op)} \\
\mid \ & x := a && \text{(assignment)} \\
\mid \ & S; S && \text{(sequence)} \\
\mid \ & \texttt{if } B \texttt{ then } S \texttt{ else } S && \text{(conditional)} \\
\mid \ & \texttt{while } B \texttt{ do } S && \text{(loop)}
\end{aligned}
$$

State: Function from variable names to values

- $\texttt{State} = \texttt{String} \to \mathbb{N}$

WHILE in Lean

```
inductive Stmt : Type where
  | skip       : Stmt
  | assign     : String → (State → ℕ) → Stmt
  | seq        : Stmt → Stmt → Stmt
  | ifThenElse : (State → Prop) → Stmt → Stmt → Stmt
  | whileDo    : (State → Prop) → Stmt → Stmt

infixr:90 "; " => Stmt.seq
```

Design choice: Shallow embedding

- Expressions are Lean functions: `State → ℕ`
- Conditions are predicates: `State → Prop`
- Simpler than deep embedding (ASTs)

Example WHILE Program

```
-- while x > y do
--   skip;
--   x := x - 1

def sillyLoop : Stmt :=
  Stmt.whileDo (fun s => s "x" > s "y")
    (Stmt.skip;
     Stmt.assign "x" (fun s => s "x" - 1))
```

What it does:

- While $x > y$
- Decrement x
- Eventually $x \leq y$

Two Kinds of Operational Semantics

Big-step semantics (natural semantics):

- Judgment: $(S, s) \Rightarrow t$
- "Starting in state s, executing S terminates in state t"
- Direct: one step from start to finish

Small-step semantics (structural operational semantics):

- Judgment: $(S, s) \Rightarrow (S', s')$
- "One step of execution"
- Execution is a chain of steps

Both are useful! Different strengths and weaknesses.

Section 11

Optional: Operational Semantics

Subsection 11.1

Big-Step Semantics

Big-Step: The Idea

Judgment form: $(S, s) \Rightarrow t$

Read as:

- Starting in state s
- Executing statement S
- Terminates in state t

Example:

$$(x := x + y;\ y := 0,\ [x \mapsto 3, y \mapsto 5]) \Rightarrow [x \mapsto 8, y \mapsto 0]$$

Single judgment for entire execution!

Big-Step: Skip Rule

$$\frac{}{(\texttt{skip}, s) \Rightarrow s}$$

Skip does nothing:

- No premises (axiom)
- State unchanged
- Simplest rule

Big-Step: Assignment Rule

$$\frac{}{(x := a, s) \Rightarrow s[x \mapsto s(a)]}$$

Assignment updates state:

- Evaluate expression a in state s
- Update state: x maps to new value
- Notation: $s[x \mapsto v]$ means state with x updated to v

Example: $(x := 5, [x \mapsto 3]) \Rightarrow [x \mapsto 5]$

Big-Step: Sequence Rule

$$\frac{(S, s) \Rightarrow t \quad (T, t) \Rightarrow u}{(S;\, T, s) \Rightarrow u}$$

Sequential composition:

- Execute S first: $s \to t$
- Then execute T: $t \to u$
- Final state: u

Intermediate state t threads through!

Big-Step: Conditional Rules

$$\frac{(S, s) \Rightarrow t}{(\texttt{if } B \texttt{ then } S \texttt{ else } T, s) \Rightarrow t} \text{ if } s(B) \text{ is true}$$

$$\frac{(T, s) \Rightarrow t}{(\texttt{if } B \texttt{ then } S \texttt{ else } T, s) \Rightarrow t} \text{ if } s(B) \text{ is false}$$

Two rules depending on condition:

- If B true: execute then-branch
- If B false: execute else-branch

Big-Step: While Rules

$$\frac{(S, s) \Rightarrow t \quad (\texttt{while } B \texttt{ do } S, t) \Rightarrow u}{(\texttt{while } B \texttt{ do } S, s) \Rightarrow u} \text{ if } s(B) \text{ is true}$$

$$\frac{}{(\texttt{while } B \texttt{ do } S, s) \Rightarrow s} \text{ if } s(B) \text{ is false}$$

While loop:

- If condition false: done (state unchanged)
- If condition true: execute body, then loop again
- Recursive rule!

Big-Step in Lean

```
inductive BigStep : Stmt × State → State → Prop where
  | skip (s) :
    BigStep (Stmt.skip, s) s
  | assign (x a s) :
    BigStep (Stmt.assign x a, s) (s[x ↦ a s])
  | seq (S T s t u)
      (hS : BigStep (S, s) t)
      (hT : BigStep (T, t) u) :
    BigStep (S; T, s) u
  ...

infix:110 " => " => BigStep
```

Using Big-Step Semantics

```
theorem sillyLoop_from_1_BigStep :
    (sillyLoop, (fun _ => 0)["x" ↦ 1]) => (fun _ => 0) := by
  rw [sillyLoop]
  apply BigStep.while_true
  { simp }  -- Condition: x > y
  { apply BigStep.seq
    { apply BigStep.skip }
    { apply BigStep.assign } }
  { simp
    apply BigStep.while_false
    simp }
```

Build proof using introduction rules!

Properties of Big-Step Semantics

What can we prove?

1. **Determinism**:
   - If $(S, s) \Rightarrow t$ and $(S, s) \Rightarrow t'$
   - Then $t = t'$
   - Execution is deterministic!

2. **Program equivalence**:
   - Prove two programs equivalent
   - Same semantics in all states

Determinism of Big-Step

```
theorem BigStep_deterministic {Ss l r}
    (hl : Ss → l) (hr : Ss → r) :
    l = r := by
  induction hl generalizing r with
  | skip s =>
    cases hr with | skip => rfl
  | assign x a s =>
    cases hr with | assign => rfl
  | seq S T s l₀ l hS hT ihS ihT =>
    cases hr with
    | seq _ _ _ r₀ _ hS' hT' =>
      cases ihS hS' with | refl =>
        cases ihT hT' with | refl => rfl
  ...
```

What Big-Step Cannot Express

Limitations:

1. **Nontermination**:
   - No judgment for infinite loops
   - Can't prove `∃t, (S, s)` $\Longrightarrow$ `t` for all programs
2. **Intermediate states**:
   - Only see start and end
   - Can't reason about what happens during execution
3. **Interleaving**:
   - Can't model concurrent execution

Solution: Small-step semantics!

Section 11

Optional: Operational Semantics

Subsection 11.2

Small-Step Semantics

Small-Step: The Idea

Judgment form: $(S, s) \Rightarrow (S', s')$

Read as:

- Starting in state s
- Executing one step of S
- Leaves us in state s'
- With program S' remaining

Execution: Chain of steps

$$(S_0, s_0) \Rightarrow (S_1, s_1) \Rightarrow (S_2, s_2) \Rightarrow \cdots$$

Small-Step: Configuration

Configuration: Pair (S, s) of statement and state

Final configuration: No more steps possible

- For WHILE: Only $(\texttt{skip}, s)$ is final
- Execution terminates when we reach final configuration

Example execution:

$$\begin{aligned}
&(x := x + y;\ y := 0,\ [x \mapsto 3, y \mapsto 5]) \\
\Rightarrow\ &(\texttt{skip};\ y := 0,\ [x \mapsto 8, y \mapsto 5]) \\
\Rightarrow\ &(y := 0,\ [x \mapsto 8, y \mapsto 5]) \\
\Rightarrow\ &(\texttt{skip},\ [x \mapsto 8, y \mapsto 0])
\end{aligned}$$

Small-Step: Assignment Rule

$$\frac{}{(x := a, s) \Rightarrow (\texttt{skip}, s[x \mapsto s(a)])}$$

Assignment in one step:

- Evaluate and update state
- Leave `skip` to indicate completion
- No further evaluation needed

Small-Step: Sequence Rules

$$\frac{(S, s) \Rightarrow (S', s')}{(S;\ T, s) \Rightarrow (S';\ T, s')}$$

$$\frac{}{(\texttt{skip};\ S, s) \Rightarrow (S, s)}$$

Two rules:

- If first statement can step: step it
- If first statement is `skip`: discard it

Second statement stays unchanged until first completes!

Small-Step: Conditional Rules

$$\frac{}{(\texttt{if } B \texttt{ then } S \texttt{ else } T, s) \Rightarrow (S, s)} \text{ if } s(B) \text{ is true}$$

$$\frac{}{(\texttt{if } B \texttt{ then } S \texttt{ else } T, s) \Rightarrow (T, s)} \text{ if } s(B) \text{ is false}$$

One step to choose branch:

- Evaluate condition
- Replace with chosen branch
- State unchanged

Small-Step: While Rule

$$\frac{}{(\texttt{while } B \texttt{ do } S, s) \Rightarrow (\texttt{if } B \texttt{ then } (S; \texttt{while } B \texttt{ do } S) \texttt{ else skip}, s)}$$

Unfold while loop:

- Replace with conditional
- If true: body then loop again
- If false: `skip` (done)
- No evaluation here - just transformation

Note: There's no rule for `skip`! (It's final)

Small-Step in Lean

```
inductive SmallStep :
    Stmt × State → Stmt × State → Prop where
  | assign (x a s) :
    SmallStep (Stmt.assign x a, s)
              (Stmt.skip, s[x ↦ a s])
  | seq_step (S S' T s s')
      (hS : SmallStep (S, s) (S', s')) :
    SmallStep (S; T, s) (S'; T, s')
  | seq_skip (T s) :
    SmallStep (Stmt.skip; T, s) (T, s)
  ...

infixr:100 " ⇒ " => SmallStep
```

Reflexive Transitive Closure

Need to chain steps together!

Use Star (reflexive transitive closure):

- $(S, s) \Rightarrow^* (S', s')$
- Zero or more steps from (S, s) to (S', s')

Big-step via small-step:

$$(S, s) \Rightarrow t \iff (S, s) \Rightarrow^* (\texttt{skip}, t)$$

This connects the two semantics!

Properties of Small-Step Semantics

What can we prove?

1. **Determinism**:
   - Each configuration steps to at most one next configuration
   - Execution is deterministic

2. **Finality**:
   - Only `skip` is final
   - Ensures we have all necessary rules

3. **Equivalence with big-step**:
   - $(S, s) \Rightarrow t \iff (S, s) \Rightarrow^* (\texttt{skip}, t)$

Finality Theorem

```
theorem SmallStep_final (S s) :
    (¬ ∃T t, (S, s) ⇒ (T, t)) ↔ S = Stmt.skip := by
  induction S with
  | skip =>
      simp
      intros T t hstep
      cases hstep
  | assign x a =>
      simp
      exact (_, _, SmallStep.assign ..)
  | seq S T ihS ihT => ...
  ...
```

Proof by induction on statement structure

Big-Step vs Small-Step

Big-step advantages:

- Simpler (one relation)
- Direct connection to result
- Easier for some proofs

Small-step advantages:

- Can express nontermination
- Can reason about intermediate states
- Can model concurrency/interleaving
- More compositional

Use whichever fits your needs!