Lean-Intro-Topology

John Rafael Matteo Munoz-Grenier

April 30, 2024

Abstract

This paper describes my senior honors thesis project: an interactive and thoroughly commented Lean code library designed to teach undergraduate students the basics of formal proof-writing, set theory, and point-set topology. This paper will motivate the need for such a library, explain its structure, and describe the lessons learned throughout the course of the project. By the end of this paper, the reader should have an understanding of how Lean works and its potential utility in mathematics education.

Contents

Chapter 1

Introduction

Many undergraduate Mathematics programs require students to take a proof-writing course, as proofs insert formal rigor into the study of Mathematics. Still, there's some degree of uncertainty, since human error is involved in evaluating the proofs and determining their consistency. For simpler proofs, the risk of human error is a lesser problem, since a discerning eye can quickly catch holes in a faulty proof. Nonetheless, grading the quality of a slew of proofs is time-consuming, especially when factoring in the time a grader might take to provide feedback for the specific errors in a poorly-written proof. Students may also have difficulty determining how much detail is sufficient for the proofs they write, if they even notice the holes in their proofs. For both instructors and students, the ambiguity in what constitutes a "formal proof" adds work outside of the mathematical and logical focus of the course.

Theorem provers and proof assistants can help resolve this ambiguity, using the determinism of computing. Theorem provers verify that the proofs encoded in the prover are sound, namely that they follow from the axioms and rules of inference. The rigorous framework of formal mathematics is thus supported by automation which can, for students, provide immediate feedback, and for instructors, grade instantly and without bias.

One such theorem prover, Lean, uses dependent type theory and the Curry-Howard isomorphism to build a programming language capable of defining and proving theorems in Mathematics [1]. Lean stands out when compared to other theorem provers like Coq and Isabelle due to Lean's extensive math library, Mathlib, and its large active community of users. Lean also has a "tactic mode," wherein all of the premises and goals of a proof are displayed as they develop throughout the proof, and functions called "tactics" can be used to advance through the proofs using automation and type inference. Lean's large user base and committed development team have also created a variety of supplemental resources for new users to learn Lean and for experienced users to reference, most notable of which

is *Mathematics In Lean* [2].

Mathematics In Lean (MIL) was the primary resource I used to learn Lean, and I found the textbook to be a smooth onramp for a student like myself with a few years of undergraduate math and computer science education. However, I found the textbook significantly harder to use once the mathematics being introduced expanded beyond what I had already studied. For an audience without much (or any) proof-based mathematics experience, MIL would be far too challenging for a first introduction to formal proof. As an alternative resource, I have developed a throroughly-commented interactive code library designed to teach undergraduate Mathematics students the basics of formal proof-writing, set theory, and topology with the assistance of Lean [3]. The code library covers first-order logic, introductory set theory, order and equivalence relations, induction, and some point-set topology.

The section on first-order logic introduces propositions, logical connectives, predicates, quantifiers, and just enough type theory to begin working with Lean. The following section on set theory introduces sets, set operations like union and intersection, functions, and cartesian products.

The section on relations considers two kinds of relations: order relations and equivalence relations. The order relations discussed are preorder, partial order, lexicographic order, and strict order. The introduction to equivalence relations transitions to equivalence classes, set/type partitions, and quotients. The section on induction brings in weak and strong induction, as well as inductive types and recursive definitions.

The final section on point-set topology introduces the definition of a topology, a topological basis, and open/closed sets. Then the section pivots to the order, subspace, and product topologies.

This code library was my honors thesis project, and this paper will explain the structure of the code library, the process of developing it, and the conclusions I came to as a result.

Chapter 2

Structure

A brief preface to the deluge of information about Lean which follows: all of the information about how Lean works comes from the Lean resources *Theorem Proving in Lean 4* [1] and *Mathematics in Lean*[2], or from my synthesis of select sections of the Lean library Mathlib, or from the experience I have gathered working with Lean for the past year.

2.1 Introduction to Lean and Logic

Conventionally, Propositional Logic is taught with a few basic ideas: A proposition is any statement which can be binarily assigned true or false, and every proposition must be either true or false. Next introduced are truth tables, and different means of combining propositions into larger propositions. We define the meaning of conjunction by appealing to a truth table, wherein `P ∧ Q` is true if and only if `P` is true and `Q` is true. Similar constructions define negation, disjunction, and implication. This proof-by-truth-table perspective is used when first learning formal logic, but is swiftly discarded as one proceeds further into mathematics. Already by the time students learn about the universal quantifier, the truth table is obsolete. A conventional foundation for mathematics also uses sets as the fundamental building block, constructing the natural numbers and so forth from the axioms of Zermelo-Fraenkel set theory [4].

Lean, in contrast, builds its whole world, even its logic, from types. In Lean, every term has some Type, and true-or-false claims have the type `Prop`, which is short for Propositions. Any given term `P` with type `Prop` (denoted `P : Prop`) is itself a type, and a term `hp` of type `P` (denoted `hp : P`) is understood to be a proof of the proposition `P`. Therefore any function which produces as its output a term of type `P` is a means by which `P` can be proven. This gives us a vehicle by which we can conceive of implication! If there is a function `f` which takes as its argument some term `hp` of type `P` and returns a term of type

`Q`, then `f` is a term with type `P → Q`. Provided that `P` and `Q` are terms with type `Prop`, `f` can be throught of as a proof that `P` implies `Q`. This means of creating implication as a function from one Proposition to another is known as the Curry-Howard Isomorphism, and provides the basis for Lean as a proof assistant.

The above mentioned introductions to logic are very different, so I needed to make a choice with my code library: Do I adapt truth tables to Lean and teach basic logic that way, or do I introduce basic logic through introduction and elimination rules, as is infitting with how Lean itself is organized? I chose to follow the path tread by the structure of Lean itself, especially because actual mathematical proof typically abandons truth tables before long. Therefore my code library starts with a brief explanation of propositions and proofs, where proofs are functions from one proposition to another. I then introduce tactics and tactic states, the other main feature of Lean. Tactics make proof-writing in Lean flow more smoothly and resemble proof-writing in Human language more than pure functional code. The tactic state also represents to the reader/writer of Lean code the names of all hypotheses already known (the function arguments and local variables) at any stage in the proof, as well as all the goals of the proofs (the type which the function should return). Each new tactic employed updates the tactic state, so proving a theorem with tactics amounts to writing tactics until all goals in the tactic state are resolved. I begin by equipping students with knowledge of just 3 basic tactics: `intro`, `apply`, and `exact`.

The `intro` tactic functions similarly to "let" in a human language proof. In some proof about Natural numbers, one might write "let n be a natural number." Similarly, in Lean, one would write `intro n`. If the current goal in the tactic state is in the form `P → Q`, then `intro hP` would update the tactic state so that there's a new hypothesis `hP : P` and a simplified goal `⊢ Q`. The chosen hypothesis name `hP` is an arbitrary name here, whatever sequence of alphanumeric characters follows the whitespace after `intro` will be the name assigned to the term introduced.

The `exact` tactic is used for finishing off a proof, and might be compared to the phrase "Because ____, the proof is complete." If the current goal is `⊢ P` and there is some hypothesis `hp : P`, then writing `exact hp` would complete the proof.

The `apply` tactic uses implications to change the goal. For example, given the goal `⊢ Q` and the hypothesis `h : P → Q`, writing `apply h` would transform the tactic state such that the new goal is `⊢ P`. Since it's known, thanks to `h`, that `P` implies `Q`, then proving `P` would suffice to prove `Q`, and the `apply` tactic packages that logic into a single line.

Before we can move on to talk about logic besides implication, there's one more facet of implication to mention: Currying. Without yet defining conjuction, we can still write types which have more than one input; If I want to create a function with two inputs and only one output, like addition on the Natural numbers (which are encoded by the type `Nat`

`:  Type`), then my function "add" will have type `Nat → (Nat → Nat)`. This type can be conceptualized by filling out the arguments of the multivariable function one at a time, so `add :  Nat → (Nat → Nat)` is a function which takes some `Nat` like `3 :  Nat` to `add 3 :  Nat → Nat`. Here, `add 3` is the function from natural numbers to natural numbers which adds three to its input. Then `add 3 6 :  Nat` is just the natural number which should be returned from 3+6, which reduces to 9. This technique is called "currying", and is critical to understanding the next section.

The code library then works through `True`, `False`, `Not`, `And`, `Or`, and `Iff`.

`True : Prop` is the proposition which is always correct, so we should always be able to produce a proof of `True`. In Lean, the proof of `True` is `True.intro :  True`. However, there's not much one can do with a proof of `True`.

`False : Prop` is the opposite, the proposition which is never correct. Thus there is not introduction rule to prove `False`, but there is the principle "ex falso", the idea that anything can follow from false premises. Lean is equipped with `False.elim :  False → P`, where `P : Prop` can be any proposition.

`Not : Prop → Prop` is denoted ¬, so for some `P : Prop`, ¬`P` is also a `Prop`. `Not P` is defined as `P → False`.

`And : Prop → Prop → Prop` is denoted ∧, so for `P Q : Prop`, then `P ∧ Q : Prop` is shorthand for `And P Q : Prop`. Unlike `True` and `False` which only had one or the other, `And` has both introduction and elimination rules:

- `And.intro :  P → Q → P ∧ Q`, so for proofs `hp :  P` and `hq :  Q`, then `And.intro hp hq :  P ∧ Q`.
- `And.left :  P ∧ Q → P`
- `And.right :  P ∧ Q → Q`

`Or : Prop → Prop → Prop` is denoted ∨, so for `P Q : Prop`, then `P ∨ Q : Prop` is shorthand for `Or P Q : Prop`. Similar to `And`, `Or` has three total introduction and elimination rules:

- `Or.inl :  P → P ∨ Q`,
- `Or.inr :  Q → P ∨ Q`, and
- `Or.elim :  P ∨ Q → (P → R) → (Q → R) → R`. This elimination rule follows from the principle that to prove any proposition `R` from a disjunction, one needs to prove that each part of the disjunction implies `R` independently.

`Iff : Prop → Prop → Prop` is denoted ↔, so for `P Q : Prop`, then `P ↔ Q : Prop` is shorthand for `Iff P Q : Prop`. The logical connective `Iff` represents bi-

implication, so `P ↔ Q` can be thought of as `(P → Q) ∧ (Q → P)`, and is codified by the introduction and elimination rules:

- `Iff.intro : (P → Q) → (Q → P) → P ↔ Q`, so for proofs `hp : P` and `hq : Q`, then `Iff.intro hp hq : P ↔ Q`.
- `Iff.mp : P ↔ Q → P → Q`
- `Iff.mpr : P ↔ Q → Q → P`

2.1.1 Dependent Types and Qualifiers

Lean implements "Dependent Type Theory," which allows for the quantifiers ∀ and ∃ to be represented in Lean. If α is some type and a predicate `p` has type α `→ Prop`, then

- `∀ x : α, p x` is the proposition that every term of type α satisifes `p`, and
- `∃ x : α, p x` is the proposition that at least one term of type α satisfies `p`.

∀ is a dependent function, so not just the output, but **the type of the output** depends on the input of the function; a term of type `∃ x : α, p x` is a function from a term `x` of type α to a proof of `p x`. In Lean, `Exists` has type `(α → Prop) → Prop`. The introduction rule `Exists.intro` has dependent type, so for any `x : α`, then `Exists.intro x` has type `p x → Exists p`. The elimination rule `Exists.elim` has type `(∃ x, p x) → (∀ a : α, p a → b) → b`, essentially saying that if `b` is implied by any term which satisfies `p`, then `b` is implied by the existence of a term which satisfies `p`.

2.1.2 Are "to be" and "not to be" the only options?

Using the the Curry-Howard Isomorphism and defining logical connectives with introduction and elimination rules creates a system of logic which is almost as expressive as classical logic, but it has a few limitations. It's not possible to prove `P ∨¬ P` for an arbitrary proposition `P` with constructive logic, the logic we have been using in Lean thus far. Lean introduces three axioms in order to use the law of the excluded middle: propositional extensionality, functional extensionality, and an axiom of choice. Propositional extensionality is the axiom that equivalent propositions are entirely equal, functional extensionality is an axiom stating any two functions which return the same outputs for any given input are entirely equal, and the choice axiom is a function of which produce any term of an arbitrary type α given that α is a nonempty type. Then using Diaconescu's theorem, it's possible to show that for any proposition `P`, either `P` or `¬P` is true.

My code library doesn't dwell very long on the differences between constructive and classical logic, nor does it discuss the measures taken by the Lean library to extend its expressive

capabilities to cover classical logic. The library is geared towards an audience of undergradute mathematics majors, so most of the time is spent developing their understanding of classical logic.

2.2 Set Theory

The structure of the Set Theory section of my code library is modeled after *Topology* by Munkres [5] and *An Introduction to Proof through Real Analysis* by Madden and Aubrey [4], along with some influence from the undergraduate course I took at the University of Arizona in Formal Proof-Writing, Math 323. My code library begins with a discussion of basic set theory, then has subsections for Relations and Induction. Induction is introduced here rather than in the opening logic section of the library because it requires more type theory to understand how induction is implemented in Lean. As students work through my code library for set theory, they will also be learning bits and pieces of type theory and how type theory is used in Lean.

2.2.1 Fundamentals of Set Theory

The first section of the Set Theory chapter in the code library is a speedy introduction to Set Theory and a referral to Chapter 4 of MIL. Sets in Lean are restricted to only contain elements of the same type, so `Set Nat` is the type of all sets which contain only natural number elements. Then each individual set `S : Set X` is a collection of terms of type `X`, which can also be represented by a predicate `p : X → Prop` by considering the collection of terms which the predicate maps to `True`. Thus a map `p : X → Prop` corresponds to the set `S : Set X` given by `x : X | p x` (the notation Lean uses for a set over terms of type X which satisfy some property, in this case `p`). And given a set `S : Set X`, the corresponding predicate is a map `p : X → Prop := λ x : X ↦ x ∈ S`. This "`λ _ : _ ↦ _`" notation is used for functions in Lean, where the blank space before the colon is the input term, the blank after the colon is the type of the input term, and the blank after the map symbol is the term outputted by the function. Thus the previous expression, `p : X → Prop := λ x : X ↦ x ∈ S`, can be read "`p` is the function of type `X → Prop` which maps a term `x` of type `X` to the proposition that `x` is an element of `S`." Since sets are defined in terms of types, there is also a set which contains all terms of the given type, which Lean denotes `Set.univ` (This set corresponds to the predicate which is always true). Using `Set.univ`, one can also think of the predicate-set correspondence as that between indicator function and the sets they indicate.

After providing readers with enough set theory background to go read MIL chapter 4, the code library provides several illustrative exercises for students to complete, some of which are adapted from MIL and some of which I created. One significant benefit of using Lean to learn mathematics is that definitions can be accessed instantly, so although MIL

doesn't define surjectivity or injectivity, students can jump to their definitions in Lean. Two of the problems which I wrote explicitly require proving that a function is injective or surjective, and students can decompose the terms using the tactic `dsimp[Surjective]` or `dsimp[Injective]`, which unfold the corresponding definitions.

The set theory also covers Cartesian products and arbitrary unions, since both will be critical for the topology chapter to come.

2.2.2 Relations

Relations are implemented in Lean as maps `r : X → X → Prop`. The main relations discussed in this section were order and equivalence relations, beginning with an introduction to bundling in Lean. Many objects in mathematics are characterized not just by the sets or functions themselves, but also by the properties they satisfy. For example, an equivalence relation is not just any relation on a type, but a relation which is also reflexive, symmetric, and transitive. Thus the object of an equivalence relation is really a quadruple: the relation itself and 3 properties of that relation. Lean accomplishes this bundling of several types into a single type with structures and typeclasses. A structure in Lean is a type with a constructor which has all the fields as arguments, and elimination functions which extract the individual fields.

```
structure <name> <parameters> <parent-structures> where
<constructor> :: <fields>
```

For example, consider how the `Equivalence` relation is expressed in Mathlib:

```
structure Equivalence {α : Type} (r : α → α → Prop) : Prop where
refl  : ∀ x, r x x
symm  : ∀ {x y}, r x y → r y x
trans : ∀ {x y z}, r x y → r y z → r x z
```

Thus the structure `Equivalence` is a bundle of 3 predicates on a given relation `r`: reflexivity, symmetry, and transitivity.

Lean also has a more sophisticated system of **typeclasses**, which are structures around which Lean has automation infrastructure. Typeclasses use almost the same syntax as structures, aside from the keyword `class` in place of `structure`. The facet of typeclasses explained in the relations section of the code library is *typeclass inference*, which is the mechanism Lean uses to infer instances of typeclasses without the programmer explicitly providing the instances. Lean keeps track of the most recent scoped definition of each particular instance, so instances don't have to be described with specific variable names to be used.

My section on relations investigates two main types of relations: **order relations** like preorders, partial orders, well-ordering, and linear orders; and **equivalence relations**.

Although there is plenty of mathematics to be done with other relations, the aim of the code library is to provide students with a sufficient mathematical foundation to begin exploring topology. The order topology and the quotient topology are essential to the study of basic topology, so I describe only the requisite information about relations for those studies. Since the order topology relies on intervals and rays in a linear order, the code library explores the orders which are weaker than linear ordering, but sufficient for intervals and suprema. Similarly, the quotient topology requires a definition of a quotient, so the code library introduces equivalence relations and builds up to how quotients are defined in Lean.

The subsection on order relations introduces readers to the strict order, then preorders and partial orders. Lean defines the set of upper and lower bounds of a set using only the preorder, but I chose to work with the confines of a partial order so that the bounded sets have a unique supremum and infimum. Then I introduce readers to the lexicographic order, which will be useful later in topology. The next file on order relations, `P3_intervals.lean`, delves into open and closed intervals and rays. Lean defines 8 different intervals (which also include rays) : `Ioo`, `Ioc`, `Ioi`, `Ico`, `Icc`, `Ici`, `Iio`, and `Iic`. The 3-letter names can be thought of as acronyms where the first word is "interval", and the second and third words are from the set "open", "closed", "infinite". The second word of the acronym describes the lower bound of the interval (or ray) and the third word describes the upper bound. The bound is a strict inequality, a non-strict inequality, or nonexistent for "open," "closed," and "infinite" respectively. This is why there are only 8 intervals described, as the interval which has no upper or lower bound is just the entire set. The interval file then introduces the reader to immediate successors and predecessors defined in terms of intervals.

The final subsection is dedicated to the equivalence relation, and is split into two files: `P4_equiv.lean` explaining the equivalence relation and `P5_quotient.lean` explaining the quotient construction. Lean has a structure `Equivalence` which was shown above, and the code library introduces the reader to that structure first, showing an example of a relation which has all three properties:

```
def parity (x y : ℤ) : Prop := ∃ k, x - y = 2 * k
```

I then introduce readers to the typeclass `Setoid`, which allows for the use of the infix $\approx$ and will be needed later for quotients. (This notation allows one to write `x ≈ y` rather than `parity x y`, for example.) The remainder of the file shows students how equivalence classes are defined and describes how equivalence classes are type partitions. Knowing now that equivalence classes subdivide a type, the next file smoothly introduces the `Quotient` type, where the terms are equivalence classes.

Lean actually has two quotient types, `Quot` and `Quotient`, where the latter is built atop the former. `Quot` is a strange kind of quotient which is built from **any** relation, so the relation need not be an equivalence. The traditional quotient is built in Lean

as `Quotient : Setoid α → Type`, so a new `Quotient` type can be created by providing an instance of `Setoid α` for some type α. The file then explains how to use this new type and how the quotient type relates to the equivalence relation on the base type, specifically by leading readers through a (mostly) worked example, the integers modulo 3. I first describe a relation `eqv (a b : ℤ) : Prop := ∃ k, a-b = 3*k`, then actually **prove** that `eqv` is an equivalence relation. From there, I construct an instance `ZSetoid : Setoid ℤ` (which is the integers $\mathbb{Z}$ with a defined relation `eqv`) and define the quotient type `Zmod3 := Quotient ZSetoid`, which consists of equivalence clasees. Since the terms of `Zmod3` are equivalence classes of integers, Lean should provide a means of constructing a term explicitly by providing an integer, and Leans does this with the function `Quotient.mk (s : Setoid α) (a : α) : Quotient s`. Lean also supplies the following useful tools for interacting with the quotient type (and many more not listed here) : `Quotient.exists_rep` posits that every term of the quotient type is equal to the equivalence class of some term in the base type, `Quotient.eq` states that two terms of the base type are *equivalent* if and only if the equivalence classes of those terms are *equal* in the quotient type, `Quotient.lift` creates a function from the quotient type to any other type β when provided some function from the base type to β which maps equivalent terms to the same output, and `Quotient.lift_mk` asserts that the function created from `Quotient.lift` maps the equivalence class to the same term that the original function would map any representative of that equivalence class. With these theorems and some more, I guide readers through a proof that no perfect square of integers is two more than any multiple of three.

2.2.3 Induction

Induction is a fundamental idea to Lean, since the language makes uses of "inductive types." Proof by induction is also a critical idea for aspiring mathematicians to learn, as it's a frequently used proof technique. This section of the code library first introduces readers to the mathematical notions of strong and weak induction, then refers readers to *Mathematics in Lean*, before returning with examples of induction and recursive definitions. Chapter 5 Section 2 of MIL was indispensible to my understanding of inductive types in Lean, and is accessibly-written, enough for the undergraduate audience of this code library, so I decided to send readers to that text directly rather than novicely rewriting the source material.

The code library just builds atop the explanation in MIL, with more explicit examples and exercises to illustrate and practice the use of induction, inductive types, and recursive definitions. There is also a brief example which proves the one missing step from a proof in the previous Quotients subsection.

2.3 Topology

This chapter of the code library is modeled more explicitly after *Topology* by Munkres, but also includes some influence from *Topologies and Uniformities* by James [6], because the Lean library Mathlib uses *Topologies and Uniformities*. This final section introduces the idea of a topology, a topological basis, and a handful of methods for constructing topologies. Unlike the primary resource for topology in Lean, Chapter 9 of *Mathematics in Lean*, this section places minimal focus on filters and metric spaces. In fact, this code library bears no mention of metric spaces, which ought to be remedied should the code library be expanded to cover more of topology. The framework for topology in Mathlib uses filters extensively, which are useful for codifying limits and continuity at a point, but I decided the filter approach to topology is less intuitive than the basis approach for a first introduction to an undergraduate student.

2.3.1 Topology and Bases

`P1_topology.lean` introduces the formal definitions of a topology and a topological basis, then scaffolds a proof that the collection of arbitrary unions of basis sets forms a topology. Just these two definitions require knowledge of much that was introduced in the previous set theory chapter: structures, typeclasses, the axiom of choice, and arbitrary unions. Lean uses a typeclass `TopologicalSpace` to encode topologies, which can make comparing two or more topologies difficult, since instances usually go unnamed in Mathlib. Thus to compare a set under two different topologies requires explicitly providing variable names for each topology into whichever theorems from Mathlib are used, otherwise Lean will infer the most recently defined instance of a suitable topology as an argument. Despite this mild annoyance, the typeclass framework is much more suitable in the common context where a set has a standard topology which would be irritating to explicitly derive at each stage. The choice to represent topologies with typeclasses also eases the use of hierarchies, so Hausdorff spaces and Metric spaces can simply be defined atop the `TopologicalSpace` typeclass.

`P2_open.lean` expands upon the definition of "open" and "closed" sets, demonstrates that the finite union and arbitrary intersection of closed sets remain closed, and introduces the definitions asnd some properties of the interior, closure, and boundary of a set.

The remaining three files, `P3_product.lean`, `P4_order.lean`, and `P5_subspace.lean` respectively cover the product, order, and subspace topologies as they are implemented in Lean.

Chapter 3

Development Process

This honors thesis project began in June 2023, stemming from an idea Dr. Sergey Cherkis had for automating the grading process in his topology class. Over the summer, I endeavored to learn Lean and become comfortable using Mathlib. In the Fall, I began building the library, and I finished in the Spring of 2024.

3.1 Summer 2023

Lean has many resources available to newcomers, so I began by tackling the introductory textbooks *Mathematics In Lean* [2] and *Theorem Proving In Lean* [1]. MIL is peppered with exercises for the reader, which I dutifully completed. When I first read through MIL, it was still written for Lean3, and I only covered the sections introducing Lean itself, set theory, and topology. When MIL was updated for Lean 4 in August, I re-read the sections of the text which were substantially (not just syntactically) changed from the previous version.

The topology section of Mathematics in Lean was unlike any other introduction to topology I had ever seen, as it took an approach to Topology centered around filters rather than bases. The filter construction of topology works well for formalization because there are fewer cases to consider, particularly in the realm of limits. Just looking at limits of a composition of two functions between metric spaces, according to MIL, there are 512 different cases. This plurality of slightly different cases poses little problem for an informal system, since one can demonstrate a particular case and hand-wave that the other cases are similar. For formalization, either 512 cases must be encoded in Lean, or another system is required, so Mathlib uses filters. The remainder of MIL's topology section is dedicated to metric spaces and topological spaces, but both are expressed entirely in the language of filters.

Having not been introduced to filters previously, I found the topology section of MIL very difficult to learn from. The text seems to presuppose knowledge of the underlying mathematics in its readership, and as such makes little effort to introduce the mathematical ideas, only taking care to delicately explain how those ideas are used in Lean. I had also just taken a semester-long course in topology, so I was particularly taken aback that the only resource for topology in Lean was so unapproachable. This became the inspirition for a new direction with my honors thesis project: writing a code library for formalizing topology in Lean which is more accessible, and based on a conventional approach like that of Munkres' *Topology* [5].

During the summer, I also read Theorem Proving in Lean 3, and then skimmed Theorem Proving in Lean 4 when it was released later in the summer to catch any changes. Compared to MIL, Theorem Proving in Lean (TPiL) focuses more on the basic constructions of Lean as a programming language, particularly on the type theory used to formalize mathematics. My impression upon reading both texts was that MIL expects more mathematical background knowledge from the reader and TPiL expects more programming savviness. I also read a bit of *Functional Programming in Lean* [7], but ultimately sidelined that textbook because it was less relevant to my goals for this project.

3.2 Fall 2023

In Late August, I began to work through the Munkres textbook *Topological Spaces* with a focus on identifying which parts of the textbook were more or less challenging to formalize. Starting in September, I pivoted to working on the first section of my instructional repository, Logic in Lean. I spent the first few weeks of September deciding how to structure the introduction to formal logic, which ultimately culminated in the current path starting with Propositions, truth and falsity, then leading through implication, disjunction, conjunction, negation, and the 2 quantifiers. I actually began writing code and comments in late September, and by mid-October I had written files explaining proofs, proposition, and implication; true, false, introduction rules, and elimination rules; negation; conjunction and disjunction; and the existential and universal quantifiers. This was also the time when I began backing my files up on Github. The remainder of the semester was spent continuing to flesh out the library, adding a section on set theory and a section on Topology.

3.3 Spring 2024

Over the Winter break, it was confirmed by the University of Arizona mathematics department that Dr. Cherkis would teach a graduate class in the coming spring, Math 529: Proof Writing and Proof Checking with a Computer. Furthermore, I was accepted as a Teaching Assistant for this class. This class gave me the opportunity to discern how approachable formalization of Mathematics in Lean is in an actual classroom environment. During this

semester, I also began writing this Honors thesis, and I finished working on the code library which introduces students to Lean.

3.3.1 Finishing the library

My original plan for the project included a subsection in the Set Theory chapter of the library for cardinality which would cover the explicit definitions of finiteness, countability, and uncountability, but after several weeks of work on the code without any forward progress, I decided to cut cardinality from the code library. The first roadblock I encountered trying to formalize Munkres' approach to cardinality was a paralysis of possibility, since Lean has finiteness defined for sets and types, each of which are useful in select situations. Mathlib has the type `Finset`, which is built atop Lean's programming infrastructure for lists, so all terms of type `Finset` are constructed explicitly by enumerating their elements. This extensional finite set construction makes proving theorems about cardinality all but trivial, but requires some type of external means to interpret intensionally defined sets as `Finsets`. Mathlib also has the type `Fin : Nat → Type` and the predicate `Finite : Type → Prop`. For a natural number `n`, `Fin n` is a subtype of `Nat` consisting of all natural numbers less than `n`. `Finite` is defined in terms of `Fin`, where a type is `Finite` if there exists a bijective function from that type to some `Fin n`. This approach to finiteness matches the formal approach described by Munkres, but requires introducing extraneous concepts from the Mathlib library used for the construction of `Finite`, namely `Equiv` and `Subtype`. I attempted writing some files in the code library relying on each approach, since `Finset` is actually used later on in topology where finite subcovers and finite intersections are concerned, but only `Finite` easily lends itself to the formal proofs of the uniqueness of cardinality and so on. `Finite` is also most similar to how countability and uncountability are defined in Mathlib, so it provides a better segue. However, I was met with another significant hurdle when trying to prove that cardinality is unique using `Finite`. Extending a function's domain to a larger type requires using Lean's if-then-else logic, which is tricky to use in tactic proofs. Even if the if-condition is met by one of the hypotheses, some elbow grease is necessary to convince Lean to simplify the if-then-else expression. This felt like an unreasonable onus to place upon students learning Lean and set theory for the first time, so I spent several hours searching for a workaround, but to no avail.

Ultimately, I decided to scrap the Countability section, for the sake of producing a cohesive code library before the end of the semester. Consequently, I also needed to drop the Compactness section of the Topology chapter of the code library. Due to time pressure, I pared down the Topology chapter even further, leaving only the first section to be written. By April 2024, I had finished writing all the remaining code within this limited scope, including solutions for all the exercises within the code library.

3.3.2 Insights as a TA

Dr. Cherkis constructed a curriculum for the graduate level course which spent the first two months leading students through the first 6 chapters of *Mathematics in Lean*, then took a handful of weeks to guide students through my code library, before returning to finish MIL in April. The class met twice weekly for 75 minutes, and a typical class session spent about 20 minutes on lecture, and the remaining time dedicated to individual or paired work on students' computers with Lean exercises. During the group and individual work time, Dr. Cherkis and I helped students upon request, clarifying concepts or quirks of Lean.

Since most of my work as a TA revolved around working with students through the difficulties they encountered while learning Lean, the following few paragraphs discuss those difficulties.

As the students were beginning to learn Lean, they often stumbled with tactics, specifically the `apply` tactic, which works like so: if you have a goal `⊢Q` and a hypothesis `h:P→Q`, then writing "`apply h`" will reduce the goal to `⊢P`. Students tended to forego reducing the goal and instead wrote "`apply hP`" where they had `hP:P` as a hypothesis. This common mistake may represent that students aren't viewing known implications as propostions themselves which require proving and specifying, just like any other proposition. The `apply` tactic also encourages a sort of backwards proof, where the goal is continually reduced until the goal follows directly from the hypotheses.

Another gripe students had with Lean was the theorem naming convention, or lack thereof. Each theorem in the extensive Mathlib library has a unique name, and while most theorems use the same naming convention, that convention is not made explicit in *Mathematics in Lean*. In general, a theorem which proves `prop2` from hypothesis `prop1` is named `prop2_of_prop1`, but the exceptions are likely more numerous than the rule. Mathlib uses certain canonical abbreviations, so "less than" is always abbreviated `le` and "multiplication" is `mul`. Additionally, Lean uses dot notation for accessing fields of a structure but also for namespace scope, so `TopologicalSpace.isOpen` could be interpreted as a field within the structure `TopologicalSpace` or as a term defined within the namespace `TopologicalSpace`.

Fortunately, the students in this class were not the first to notice the difficulty of locating desired theorems and definitions, so there are several resources for searching through the Mathlib library. The website **moogle.ai**, the tactics `apply?` and `exact?`, and some other tricks can be used to ease the difficult task of finding the theorem you want. When writing Lean code in the VSCode editor, one can easily jump to the file in the library which defines a particular expression, and this is probably true of many other code editors which support Lean. The tactics `apply?` and `exact?` perform a search through the library for theorems which imply the goal or directly close the goal respectively. Lean also has keywords `#check` and `#print` which display the type and definition respectively of

whichever term follows the keyword. Still, it took some time for students to develop an intuition for how theorems in Mathlib are likely to be named, so an explicit guide would be an invaluable resource.

Of course, this trial class wasn't all hurdles and setbacks. While the stumbling blocks for students helped provide feedback for how to improve my code library, there were also many aspects of the course which functioned near seamlessly.

During the semester, Dr. Cherkis was able to set up an autograder through Gradescope where students could submit their Lean code proofs for assigned theorems, and Gradescope would give them a score according to how many of their proofs successfully compiled in Lean. Admittedly, this procedure is a bit redundant, since the extra work of submitting the code is after Lean has already successfully compiled on the student's computer. Nonetheless, this Gradescope submission process made it possible to submit completed homework and receive a grade automatically, cutting down on the work necessary for the instructor. However, there is still room for improvement, since students needed to slightly adapt their code which compiles locally to code which Gradscope can compile, and Dr. Cherkis needed to repeatedly rewrite sections of code and display files for each assignment he posted to Gradescope.

By the end of the semester, the majority of the students taking the class were self-sufficient with Lean, and could even begin to contribute to Mathlib themselves! Students also seemed to have greater clarity about the mathematics which they hard worked to formalize, as was my experience formalizing the mathematics in this code library. Some students also expressed the addictive, game-like nature of formalization in Lean which comes from the instant gratification of the compiler's feedback when a proof is completed. During my experience with Lean, I learned about a lot of mathematics which was new to me, including lattice orders, filters, uniform spaces, and type theory. The graduate students taking this course likely have more math experience and knowledge than I do, but I imagine there are some new mathematical ideas they were exposed to during the course, aside from the greater mastery of the mathematics they already understood and formalized.

3.3.3 Future Development

After the success of this graduate-level formalization course, I am hopeful for the prospects of an undergraduate-level course which follows my code library. There are a few other courses in other universities which have successfully done what this project aims to do: teach undergraduate students how to reason mathematically and write proofs using Lean. Dr. Heather Macbeth wrote an excellent online textbook, *The Mechanics of Proof* [8], which accompanies her course at Fordham University. Dr. Kevin Buzzard of Imperial College London also teaches a course on formalizing mathematics using Lean [9], which he has taught for a couple of semesters to undergraduate math students.

For my code library to be a sufficient resource for an undergraduate course, it still requires some more work; the code library needs an accompanying textbook, human language proofs, and a grading pipeline. Ideally, the code library would be less commented, and the content of the comments would be moved over to a textbook, similar to how *Mathematics in Lean* has both a pdf file and a code library. Then that textbook could also contain several proofs which are written both in Lean and in human language, so students might also learn how to write human language proofs and not just Lean proofs. Lastly, my code library needs to include an easy-to-use system for creating assignments and compiling Lean proofs for those assignments. Dr. Cherkis used Gradescope to do this for our course, but I would like a system which requires a bit less clerical work from both instructor and student.

My code library is also missing a subsection of Set theory which discusses finite, infinite, countable, and uncountable sets. The library could also reach much further into topology, covering at least continuity, compactness, and connectedness. With the hindsight of TAing this course, the introduction section of the library could also be improved by taking more time on tactics, particularly `apply` and `rewrite`.

Once my code library has been upgraded enough to be used as a reference for an undergraduate course, there are myriad possibilities for uses. Instructors at any university could use the library for their own courses in formal mathematics, and the library could be developed to focus less on topology and instead anywhere else in mathematics. In my undergraduate proofs course, topology wasn't discussed, but instead some elementary Real Analysis and Field theory.

I have also surmised from my experience working with Lean and helping others to learn Lean that there *really* ought to be a user guide for tactics which explains what all the tactics do, and that Mathlib could use some extra document which can help with searching through the library. The tools I have learned to find specific theorems in Mathlib were hard-fought, but they don't have to be.

Bibliography

[1] J. Avigad, L. de Moura, S. Kong, and S. Ullrich, *Theorem Proving in Lean 4*, 2023. [Online]. Available: https://lean-lang.org/theorem_proving_in_lean4/

[2] J. Avigad and P. Massot, *Mathematics in Lean*, 2023. [Online]. Available: https://leanprover-community.github.io/mathematics_in_lean/mathematics_in_lean.pdf

[3] R. Grenier, "Lean-intro-topology," 2024. [Online]. Available: https://github.com/charlespwd/project-title

[4] D. Madden and J. Aubrey, *An Introduction to Proof through Real Analysis*. Wiley, 2017. [Online]. Available: https://books.google.com/books?id=7kkzDwAAQBAJ

[5] J. Munkres, *Topology*, ser. Featured Titles for Topology. Prentice Hall, Incorporated, 2000. [Online]. Available: https://books.google.com/books?id=XjoZAQAAIAAJ

[6] I. James, *Topologies and Uniformities*, ser. Springer Undergraduate Mathematics Series. Springer London, 2013. [Online]. Available: https://books.google.com/books?id=k4nbBwAAQBAJ

[7] D. Christiansen, *Functional Programming in Lean*, 2023. [Online]. Available: https://lean-lang.org/functional_programming_in_lean/

[8] H. Macbeth, *The Mechanics of Proof*, 2023. [Online]. Available: https://hrmacbeth.github.io/math2001/index.html

[9] K. Buzzard, *Formalising Mathematics*, 2022. [Online]. Available: https://www.ma.imperial.ac.uk/~buzzard/xena/formalising-mathematics-2024/index.html