Honors Thesis Project

John Rafael Matteo Munoz-Grenier

April 18, 2024

Contents

Introduction

Most 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 still involved in evaluating the proofs and determining their consistency. [Insert examples of theorems which have been erroneously proven throughout history]. 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. The Lean Theorem prover offers a solution to the first problem, and this project is an effort to simplify the grading process of proofs by integrating Lean in a proof-writing course.

0.1 What is Lean?

Lean uses dependent type theory and the Curry-Howard isomorphism to build a programming language capable of defining and proving theorems in Mathematics. Although other theorem provers exist, Lean was chosen for this project due to its 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 [insert screenshots] 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.

0.2 Why use Lean?

1. Autograding! Any proof written in Lean which compiles is guaranteed to be correct.
2. Lean Infoview makes writing and reading proofs easier; the precise state of what is already proven and what remains to be shown is made explicit for every step of the

proof.

3. Tactics can automate some of the tedious parts of a proof, like unfurling definitions and other procedures which can be done algorithmically.
4. All of the essential parts of a proof are made explicit; hypotheses which go unused are automatically marked by Lean, and each use of some hypothesis or lemma is tracked by its variable name.

Downsides:

- Some parts of a proof which might be natural or trivial may be challenging to formalize.
- Appealing to some well-known lemma requires knowledge of its name in Lean.
- Lean proofs are not written like human language proofs, and don't directly develop a student's capacity to write readable human language proofs.
- Lean takes some time to learn, time which could instead be used for purely mathematical pursuits.
- Lean requires some level of tech savviness to access, since it must be downloaded and configured, and either the terminal or a code editor is necessary to write Lean code.

0.3 Who else uses Lean?

Heather Macbeth [Insert information about her course] [Insert other examples]

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.

0.4 Summer 2023

Lean has many resources available to newcomers, so I began by tackling the introductory textbooks Mathematics In Lean and Theorem Proving In Lean. Mathematics In Lean is peppered with exercises for the reader, which I dutifully completed. When I read through MIL, it was still written for Lean3, and I only covered the sections introducing Lean itself, set theory, and topology.

The topology section of Mathematics in Lean was sparse, and took an approach to Topology centered in the notion of filters.

Definition 1 *A* **Filter** *F is a collection of sets over a space X such that*

1. *if $S \in F$ and $\hat{S}$ is a superset of S, then $\hat{S} \in F$, and*
2. *for any two sets $S, T \in F$, then the intersection $S \cap T \in F$.*

0.5 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.

0.6 Spring 2023

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.

0.6.1 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 `n : Nat`, `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` 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 the limited scope, including solutions for all the exercises within the code library.

0.6.2 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.

0.6.3 Future Development

Structure

0.7 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 only if `P` is true and `Q` is true. Similar constructions define negation, disjunction, and implication.

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` is itself a type, and a term `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 use adapt truth tables into 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), 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 the tactic state represents that there are no goals left to be solved. I equip students with knowledge of just 3 basic tactics in the beginning: `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`. `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 that `P` implies `Q`, proving `P` would suffice to prove `Q`, and the `apply` tactic packages that logic into a single line.

The code library then works through `True`, `False`, `Not`, `And`, `Or`, and `Iff`. `True` and `False` are both propositions, where `True.intro` is a function which returns a proof of `True` and requires no arguments, and `False.elim` is a function which takes a proof of `False` as an input and can output a proof of any proposition. `Not` has type `Prop → Prop` and is denoted ¬, so for some `P : Prop`, `¬ P` is also a `Prop`. `Not P` is defined as `P → False`. `And`, `Or`, and `Iff` are all terms of type `Prop → Prop → Prop`, and are denoted by ∧, ∨, and ↔ respectively. These logical connectives all have introduction and elimination rules: introduction rules for creating a term with the type of the connective, and elimnation rules for turning terms with the type of the connective into proofs of other propositions. For example, `And.intro` takes proofs of `P` and `Q` and returns a proof of `P ∧ Q`. `And` has two elimination rules, `And.left` takes a proof of `P ∧ Q` to a proof of `P`, and `And.right` takes a proof of `P ∧ Q` to a proof of `Q`.

0.7.1 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 `p` has type α `→ Prop`, **TBD**

0.7.2 Are "to be" are "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 shortcomings. 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 prove 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 which produces an 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.

The 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.

0.8 Set Theory

The structure of the Set Theory section of the code library is modeled after *Topological Spaces* by James Munkres and *An Introduction to Proof through Real Analysis* by Trench, along with some influence from the undergraduate course I took at the University of Arizona in Formal Proof-Writing. The code library begins with a discussion of basic set theory, then has subsections for Relations, Induction, and Cardinality. 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 the code library for set theory, they will also be learning bits and pieces of type theory and how type theory is used in Lean.

0.8.1 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. Lean identifies sets with predicates on a type, i.e. a map `p : X → Prop` corresponds to the set `S : Set X` given by `x : X | p x`. And given a set `S : Set X`, the corresponding predicate is a map `p : X → Prop := λ x ↦ x ∈ 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` **TBD**

0.8.2 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 4 things: 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 need to 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 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 a file explaining the equivalence relation and a file 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. The remainder of the file shows students how equivalence classes are defined and describes how equivalence classes are set 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` and define the quotient type `Zmod3 := Quotient ZSetoid`. 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.

0.8.3 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.

0.9 Topology

This chapter of the code library is modeled more explicitly after *Topological Spaces* by James Munkres, but also includes some influence from *Topologies and Uniformities* 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.

0.9.1 Topology and Bases

The first file introduces the formal definitions of a topology and a topological basis, then scaffolds a proof that the collection of arbitrary unions of basis sets is 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 benefits ease of use of hierarchies, so Hausdorff spaces and Metric spaces can simply be defined atop the `TopologicalSpace` typeclass.

The second file 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 and some properties of the interior, closure, and boundary of a set.