Honors Thesis Project

John Rafael Matteo Munoz-Grenier

March 23, 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

0.6.2 Insights as a TA

For the first month and a half of the class, Dr. Cherkis lectured in parallel to the structure of Mathematics in Lean 4. I was taken aback by the difficulty students had in grasping the "apply" tactic, which transforms a tactic state with goal Q to a tactic state with goal P by using an implication $P \to Q$.

0.6.3 Honors Thesis 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 $P \wedge Q$ 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 $P \to Q$. 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 $\rightarrow$ 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 blank, 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 $\rightarrow$ 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 $Prop \rightarrow Prop$ and is denoted $\neg$, so for some Prop P, $\neg P$ is also a Prop. $Not\ P$ is defined as $P \rightarrow False$. And, Or, and Iff are all terms of Type $Prop \rightarrow Prop \rightarrow Prop$, and are denoted by $\wedge$, $\vee$, and $\leftrightarrow$ 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 $P \wedge Q$. And has two elimination rules, $And.left$ takes a proof of $P \wedge Q$ to a proof of P, and $And.right$ takes a proof of $P \wedge Q$ to a proof of Q.

0.7.1 Dependent Types and Qualifiers

Lean implements "Dependent Type Theory," which allows for the quantifiers $\forall$ and $\exists$ to be represented in Lean. If α is some type and p has type $\alpha \rightarrow Prop$, **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 $P \vee \neg P$ 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 $\neg P$ 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 $p : X \to Prop$ corresponds to the set $S : \text{Set } X$ given by $\{x : X \mid p\ x\}$. And given a set $S : \text{Set } X$, the corresponding predicate is a map $p : X \to Prop := \lambda x \mapsto x \in S$. 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" []

0.8.2 Relations

Relations are implemented in Lean as maps $r : X \to X \to Prop$. 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>
```

0.8.3 Induction

0.8.4 Cardinality

0.9 Topology

This section 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 of the code library is much less linear, with only the basics subsection as a dependency for the other subsections. The basics subsection introduces the idea of a topology, a topological basis, and a handful of methods for creating new topologies. The remainder of the topology section is devoted to specific topological properties: compactness, continuity, connectedness, and separation.

0.9.1 Topology and Bases

0.9.2 Continuity

0.9.3 Connectedness

0.9.4 Compactness