Automated Proof Verification with Lean

A Thesis Submitted to the Faculty of

Georgetown College

In Partial Fulfillment of the Requirements for the

Honors Program

By

Logan Johnson

Georgetown, Kentucky

May 2024

Abstract

Automated Proof Verification with Lean

Logan C. Johnson

Director: Dr. Homer White, Ph.D.

This thesis is meant to explore Lean Theorem Prover and address its potential in automating the verification of mathematical proofs. I explore how lean functions as a functional programming language and how this changes the proof process. This leads to Lean treating theorems as functions that essentially convert known hypotheses into a new conclusion. Lean does this by taking advantage of typed objects rather than following a traditional set theory approach. I then directly compare the ways in which Lean proofs differ from traditional paragraph-style proofs and try to explain the upsides and downsides of these differences. Lean does end up being useful for automated proof verification, so long as one is able to learn the language and get used to the different style of proof. It is still being developed and is only becoming more powerful with time, so its userbase will likely only continue to grow.

APPROVED BY THE DIRECTOR OF HONORS THESIS:

_____

Dr. Homer White, Department of Mathematics

APPROVED BY SECOND READER OF HONORS THESIS:

_____

Zachary May

APPROVED BY THE HONORS PROGRAM:

_____

Dr. Barbara Burch, Director

DATE:_____

Table of contents

Acknowledgements

Firstly, I need to mention that this entire thesis has been formatted using the book *How To Prove It With Lean,* written by Daniel J. Velleman. His book was instrumental in learning to program in Lean and without this book all of my work would have been much more difficult. I would also like to give thanks to my thesis director Dr. White for the time and effort that he has put into this project. He chose to work with me on this project over the summer and without his guidance I could not have completed something of this scale. Another round of thanks should be given to my second reader, Zachary May. Zachary May is a senior software engineer with Upbound Group, specializing in web technologies and functional programming. He received his M.S. in computer science from the University of Kentucky in 2016, with a focus on software engineering, programming language theory, and natural language processing. His feedback has been quite beneficial in getting this up to the level of quality it is at now. Finally, I would like to thank my parents for supporting me throughout all of my schooling and always taking care of me.

Preface

This thesis is intended to address the potential that the Lean programming language has in aiding with the verification of mathematical proofs. Lean was developed as a functional programming language with the intent that it could be used for theorem proving. Development on the language started in 2013 and has now seen four different fully released versions. I would recommend that while reading this thesis, you download Lean and put my code onto your own machine to see how it works. I have included a file on my github repository which contains code for each of the examples I put into the "Lean as a Theorem Prover" section of the thesis. The `Mathlib` package is not contained within this repository, so in order to see my examples for yourself, you will need to copy the examples into your own Lean project. While I have done my best to show the language in a way that displays both its positive and negative aspects, the best way to get a grasp for that will be to try it out for yourself. Instructions on how to install a copy of Lean can be found here.

If you find this language interesting and would like to learn more, the textbook How To Prove It With Lean is a great starting point, and the textbook Mathematics in Lean goes into further detail on some higher level topics such as calculus and topology. Lean also has

a dedicated forum on Zulip where other users can help address any questions regarding the language. I used the forum a few times while working on this project, and the feedback I recieved was quite helpful.

1 Functional Programming

Basic Functional Programming

A programming paradigm is a way of categorizing any given programming language based on its features and how it computes its code. Programming languages will typically fall into one main programming paradigm even if they are capable of using multiple, sometimes chosen because of the intended use of the language. Lean happens to be a functional programming language, meaning that every single thing in Lean is a function. A deep understanding of functional programming is not necessary to utilise Lean, but in order to take full advantage of the theorem proving capabilities it is able to provide, it helps to have some idea of how the language works.

When we write out mathematical proofs on paper, it is easy to think of each step we take or theorem we apply as part of a bridge between the hypotheses we are assuming to be true and the result we are trying to prove. Another way of thinking about each step in the proof process is as that of a function. In some proofs this is easier to picture than others, but when we take a step in a proof, we are essentially converting some fact that we know to be

true into a different fact that we now know to be true. This process repeats over and over again until eventually we have converted something into our end goal. When the process of proving is thought of in this way it becomes more clear how Lean is able to verify proofs.

We could think of Lean as a sort of kitchen in which we are cooking proofs out of raw hypotheses. When making a cake, we have to start with eggs, flour, milk, sugar, and a few other ingredients and take specific steps towards finishing the cake. We may first crack the eggs open before mixing them in alongside the flour and sugar. Later on we need to pour the batter into a pan and then bake that in the oven. Completing a proof is done in a somewhat similar way, as we mix hypotheses together to get closer to our goal and use different methods of proof such as contradiction or induction like an oven that cooks our hypotheses into a finished proof. Lean effectively provides a space where we can do that cooking while it verifies that we didn't skip any steps in the process along the way. Lean makes sure that we actually cracked the eggs before mixing them in just as is makes sure we are applying a theorem in exactly the way in which it needs to be applied.

One more important thing to mention before diving into an example is that Lean also takes advantages of *types.* We could say for example that 5 has type Nat, or natural. This means that every single thing in Lean has a given type, which can lead to some issues if typing is not taken care of correctly. Just as some given number in mathematics could be a natural, integer, rational, irrational, or real, that same number in Lean could be assigned to one of those specific types. While we typically think of a given natural number as also being an integer, rational, and real, Lean tends to be strict and say that each natural number is only a natural number. While this level of specificty is quite useful in maintaining logical

consistency, it can make some processes a bit more difficult. I ran into a few issues during my time with the language when trying to make a recursively defined function that took a natural number to represent which term was being computed and returned a real number. Because Lean was so strict with its typing, it did not allow me to define that recursive function the same way it would easily be defined if I was trying to output another natural number. While the theorems in Lean are functions, the actual mathematical information is stored in types. Lean is then able to logically verify each step of a proof because of this strict typing. It will not allow for one type to be changed to another type without a valid function being applied.

Theorems as Functions

Let's compare a simple Lean function to a simple theorem.

Simple Function

```
def adding (n1 : Nat) (n2 : Nat) : Nat :=

  n1 + n2
```

Simple Theorem

```
example (x : α) (A B : Set α) (h1 : A ⊆ B)
    (h2 : x ∈ A) : x ∈ B := by
  apply h1
  apply h2
  done
```

When using Lean, you set up a theorem to be proved using similar notation to the notation used when defining a function. The hypotheses you are assuming to be true are put in parentheses, just like arguments to a function. In the simple function above I defined two arguments for the function, the two natural numbers `n1` and `n2`. The function then takes these naturals and adds them together to produce another natural number. Within each set of parentheses is information separated by a colon. The information on the left side defines something, and the right side tells us what type that previously defined thing has. Whereas in the simple function we are only defining things of type natural, in the theorem we see that Lean will allow us to define objects of an arbitrary type `α` so long as we continue to use that arbitrary type throughout the rest of the theorem.

The function itself is even split into two parts, with everything left of the colon being inputs, and everything to the right being the type of the output of the function itself. When we set up a theorem or example in Lean, Lean will not let us end the theorem until we have something of the type set out when defining the theorem. This is where we are able to use tactics and apply previous theorems to either convert the types of hypotheses we are assuming, or convert the type of the goal we are trying to prove. In my example

theorem above, I applied our first hypothesis which acted as a function and changed the current type in the function. After applying the second hypothesis, I had essentially applied functions to convert our assumptions from their initial types into the goal we wanted to prove. Now that we have a proved theorem, we can apply this in proving other theorems just like any other built-in theorem. Since everything is a function, Lean is easily able to use newly proved theorems to convert the types of hypotheses in a different theorem. Having some basic understanding of how the language works, we can begin to apply these concepts to more complex theorems and see the potential benefits of using Lean.

Note: `h1` is not actually a proposition claiming that $A \subseteq B$, but rather an object of *type* $A \subseteq B$ that we call `h1`. It is also interesting to note that within Lean even the various types have their own types. The type $A \subseteq B$, for example, would have type proposition. Each of what we traditionally think of as hypotheses would be types in Lean with a type of proposition. Because Lean takes this approach with types, it means that Lean is not concerned with truth or falsehood. Lean does not necessarily prove that anything is true or false, but instead is focused on converting objects of one type into another. Because of the way in which Lean was set up, if some hypotheses can be converted from one given type to another we then have a true theorem. Despite this, the language itself is not actually claiming any given object or proposition is true or false.

2 Lean as a Theorem Prover

Differences From Paragraph Style Proofs

Despite the incredible power that Lean could provide in the verification of mathematical proofs, this does pose some difficulties, namely the ease with which the aforementioned proofs can be written up. Typically, proofs are simply written up in a paragraph style, where the steps being taken and the theorems being applied are laid out in plain terms so that it can be easily understood by fellow mathematicians. There are often times when mathematicians writing a typical proof will take things for granted or skip over steps that they think the reader will either already know to be fact or can easily reason out for themselves. This lax approach for conveying information simply does not work when trying to communicate with technology, and a much more specific and methodical approach must be adopted in order to take advantage of the logical verification benefits. Thankfully, Lean has a community working to create libraries of previously proven theorems that can be applied to speed up the writing and verification of future proofs. This thankfully means that all proofs do not need to be taken all the way back to basic axioms: Users can save

time by avoiding proving adjacent theorems and instead focus only on the immediately relevant steps of their proof.

Term Mode vs. Tactic Mode

Lean features two different ways to write out proofs: term mode and tactic mode. Writing out a proof in term mode is typically done with simpler proofs where you can simply chain together a couple theorems or hypotheses to acquire the desired goal, whereas in tactic mode Lean will break everything down into steps. All of the examples later in this thesis will be completed in tactic mode because this is where lean is really at its strongest. To go back to the earlier example of Lean being a sort of kitchen, we could think of writing a proof in term mode like making a salad. You have a few different ingredients that you lay on top of each other to make something new, but there isn't anything changing in the process of making the salad. There are also no tools being used when making the salad. You lay all of the individual pieces in a bowl and you may mix them around a bit, but there is nothing you cannot do by hand. The lettuce contributes to making the salad, but it still remains lettuce within the context of the final product. Lean being put into tactic mode however is something more akin to making a lasagna. Different tactics in Lean are like tools in a kitchen, where they can in one step convert something that you currently have into something different. When making our lasagna of a proof, we could employ tactic stove to our assumed raw beef hypothesis to produce cooked ground beef. We could also apply tactic oven to essentially cook the lasagna ahead of time and reduce our final goal to getting all of the ingredients in a pan. This ability to break each proof down step by step is

much more similar to a paragraph style proof than writing in term mode is, and the tactics it gives us are where the power of lean really comes from.

Here is a simple example of the difference between a term style proof and a tactic style proof. I won't go into too much detail since we will see the benefits of tactic mode later, but it will be useful to see other options that lean offers.

Term Mode Proof

```
example (x : α) (A B : Set α) (h1 : A ⊆ B)
    (h2 : x ∈ A) : x ∈ B := h1 h2
```

Tactic Mode Proof

```
example (x : α) (A B : Set α) (h1 : A ⊆ B)
    (h2 : x ∈ A) : x ∈ B := by
  apply h1
  apply h2
  done
```

Both examples are proving the same thing, but we can tell one is in tactic mode because of the `by` right after the `:=` which indicates the beginning of the justification for the proof. This state will now allow us to use tactics which modify our goals into something easier to reach, or modify our hypotheses into something more directly useful.

`by_cases`

One specific example of a tactic we can use when doing a proof in tactic mode is that of `by_cases.` When completing a proof where some object could have multiple possible states, if we show that each individual state would lead to the desired conclusion, then we have completed the proof. If, for example, some variable x that we are working with is an integer, then we could say that there are two possible cases: either x is even or x is odd. We could then assume first that x is even and use that to complete our proof followed by assuming that x is odd and finishing the proof. If we say that x being even is a proposition P, and we call our end goal R, then when using proof by cases it is true that if $P \Rightarrow R$ and $\neg P \Rightarrow R$, then R is true.

This is one example of a rule of inference used in mathematical proofs. Different tactics in Lean correspond to different rules of inference and will modify our current goal or one of our hypotheses. We could again think of each of theses rules as tools in our mathematical kitchen which allow us to work our current knowledge into a fully formed proof.

For each of the following proofs, I will first provide a typical "paragraph style" version of the proof, so the differences between the two can easily be compared.

Inequality Addition

Paragraph Style Proof

Theorem. *If $a < b$ and $c \leq d$, prove that $a + c < b + d$*

There are multiple ways to approach this in a paragraph style proof, so I will attempt to have this proof follow along the same lines as the Lean proof.

Proof. There are two possible cases: either $c = d$ or $c < d$. We will first consider the case where $c = d$. We know $a < b$, so it would also be true that $a + c < b + c$. Then because $c = d$, $a + c < b + d$. Now consider the case where $c < d$. We know $a < b$, so $a + c < b + c$ and $b + c < b + d$ because $c < d$. Thus by transitivity of inequalities, we could say $a + c < b + d$ □

Lean Proof

Seting up the problem

Here I put the theorem we want to prove into Lean and we can see the resulting infoview panel. I name our two assumptions `h1` and `h2`, for hypotheses one and two. After a colon I then write out the thing I am trying to prove with those hypotheses and use `by` to put Lean into tactic mode.

It can now be seen that the infoview panel lists out both of our hypotheses as well as the goal we are working towards at the bottom. This panel will continue to change as more code is added to the Lean file. One other thing to note is that the `done` at the end of the proof is underlined in red, indicating that the proof of this theorem is not entirely complete.

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
abcd: ℝ
h1: a < b
h2: c ≤ d
⊢ a + c < b + d
```

Step 1

Here I lay out the two possible cases of our second hypothesis which allows me to strengthen the information that we know. We see this strengthened hypothesis reflected in `h3` in the infoview.

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
abcd: ℝ
h1: a < b
h2: c ≤ d
h3: c = d
⊢ a + c < b + d
```

Step 2

Here I used hypothesis 3 to rewrite the `c` in our final goal as a `d`. This change is reflected in the infoview for this step. This is where we can see tactics changing our goal into something that will be easier for us to reach.

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d
  rw [h3]


  done
```

Tactic State in Infoview

```
R: Type u_1
inst✝: Ring R
abcd: ℝ
h1: a < b
h2: c ≤ d
h3: c = d
⊢ a + d < b + d
```

Step 3

In this step I applied a theorem already in the Mathlib library for Lean. The `add_lt_add_right` theorem simply states that if you have a `b < c`, then `b + a < c + a` which is exactly what we need to prove the goal for the first case. As the first case has been completed, the infoview then switches to the second case which is reflected in the new `h3` and reset goal. Because we defined `h3` earlier as the hypothesis related to the cases, Lean automatically assigns the new case to `h3` as well.

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d
  rw [h3]
  apply add_lt_add_right h1


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
abcd: ℝ
h1: a < b
h2: c ≤ d
h3: ¬c = d
⊢ a + c < b + d
```

Step 4

In order to better work with our new hypothesis, I use a tactic which pushes the negation symbol further into the thing it is negating. This results in a hypothesis which can actually be applied later on. This is a perfect example of a tactic in lean modifying one of our hypotheses, rather than just the end goal.

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d
  rw [h3]
  apply add_lt_add_right h1
  push_neg at h3


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
abcd: ℝ
h1: a < b
h2: c ≤ d
h3: c ≠ d
⊢ a + c < b + d
```

Step 5

Here I am laying out a new hypothesis which will be useful later in the proof. This hypothesis seems like an obvious conclusion based on hypotheses two and three, but we must still lay it out simply for Lean if we want to actually use it. Reaching this hypothesis is necessary in order to use the transitivity of inequalities like we do in the paragraph style proof, and proving it in this way is sort of like proving a lemma within a paragraph style proof. The infoview panel always displays the most current goal, which is why it is displaying the goal for `h4` rather than the main goal.

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d
  rw [h3]
  apply add_lt_add_right h1
  push_neg at h3
  have h4 : c < d := by


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
abcd: ℝ
h1: a < b
h2: c ≤ d
h3: c ≠ d
⊢ c < d
```

Step 6

Here I apply another theorem already in Lean which takes the information `h3` and `h2` gives us and shows our current goal. Writing out `h4` like this is technically optional, as Lean allows you to evaluate tactics within arguments for other tactics. Despite this, I personally find it more clear and human-readable to write out extra hypotheses like this rather than just giving the body of the argument when necessary. Below, I display two different infoviews that are able to be seen when at this step. What information Lean gives in the infoview depends on where the cursor is placed in the proof, with Lean only taking into account tactics used up until wherever the cursor is placed. Note that I will be using | to represent on which line the cursor is located to get the infoview state shown.

When the cursor is placed within the lemma, it will display information about the lemma.

Since we have completed the proof of this lemma, the infoview will say we have no more goals. Even though the infoview says here that we have no more goals, we can know that the larger proof is not yet completed because of the underlined `done` at the end. If we move our cursor down a line and back into the main proof the infoview will again display information on the main proof.

Lemma State

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d
  rw [h3]
  apply add_lt_add_right h1
  push_neg at h3
  have h4 : c < d := by
    apply Ne.lt_of_le h3 h2 |


  done
```

Tactic State in Infoview

```
No goals
```

Main Proof State

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d
  rw [h3]
  apply add_lt_add_right h1
  push_neg at h3
  have h4 : c < d := by
    apply Ne.lt_of_le h3 h2
  |
  done
```

Tactic State in Infoview

```
abcd: ℝ
h1: a < b
h2: c ≤ d
h3: c ≠ d
h4: c < d
⊢ a + c < b + d
```

Step 7

I now use the calc tactic to work through the rest of the theorem. This tactic is quite useful as it allows us to chain together multiple equalities or inequalities while still giving proofs for each step. This is essentially a shortcut of writing out individual hypotheses and then using the rewrite tactic to get our desired goal.

In this case, I only need to do two steps of chaining inequalities, where I use transitivity to show that the starting value is less than the final value. It essentially follows the same path as the paragraph style proof, where the tactics `add_lt_add_right` and `add_lt_add_left` justify the steps taken.

Lean File

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d
  rw [h3]
  apply add_lt_add_right h1
  push_neg at h3
  have h4 : c < d := by
    apply Ne.lt_of_le h3 h2
  exact calc
    a + c < b + c := add_lt_add_right h1 c
    _ < b + d := add_lt_add_left h4 b
  done
```

Tactic State in Infoview

```
No goals
```

a is Less Than or Equal to b

Paragraph Style Proof

Theorem. *Suppose that $a, b \in \mathbb{R}$ and for every $\varepsilon > 0$, we have $a \leq b + \varepsilon$. Show that $a \leq b$.*

Proof. Assume for the sake of contradiction that a is not less than or equal to b. Then it would be true that $a > b$. Now consider the case where $\varepsilon = \frac{a-b}{2}$. Then since $a > b$, epsilon

is positive and by our assumption then $a \leq b + \varepsilon$. Then

$$\begin{aligned} a &\leq b + \varepsilon \\ &= b + \frac{a-b}{2} \\ &= b + \frac{a}{2} - \frac{b}{2} \\ &= \frac{a}{2} + \frac{b}{2}. \end{aligned}$$

So now,

$$\begin{aligned} a &\leq \frac{a}{2} + \frac{b}{2} \\ a - \frac{a}{2} &\leq \frac{b}{2} \\ \frac{a}{2} &\leq \frac{b}{2} \\ a &\leq b. \end{aligned}$$

But now we have that $a \leq b$ and $a > b$, a contradiction! □

Lean Proof

Setting up the problem

I again set up the proof with our one hypothesis and the goal we want to prove. These are then seen listed in the infoview on the right.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
⊢ a ≤ b
```

Step 1

The `by_contra` tactic allows me to complete this problem using proof by contradiction. This tactic automatically creates a hypothesis containing the negation of the final goal named `h2`, and changes the final goal to `False` meaning that it needs us to show a contradiction.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: ¬a ≤ b
⊢ False
```

Step 2

Here I use the `push_neg` tactic similarly to the previous example to get a usable version of `h2` as well as pick a specific epsilon for which we will find a contradiction. This new epsilon will now show up in the infoview on the side and can be used in our problem.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2


  done
```

Tactic State in Infoview

```
R: Type u_1
inst✝: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: b < a
ε: ℝ := (a - b) / 2
⊢ False
```

Step 3

Here I lay out a hypothesis that we will later be able to apply to `h1`. Claiming that epsilon was positive in the paragraph style proof is fairly simple to back up, where we only really need to justify that $a - b$ is positive. In Lean however, it requires a bit more effort and as such I put it in its own hypothesis.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2
  have h3 : ε > 0 := by


    done


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: b < a
ε: ℝ := (a - b) / 2
⊢ ε > 0
```

Step 4

Anyone reading a paragraph style proof such as ours would know that dividing a number by two does not impact whether the resulting number will be positive or negative, but this still needs to be justified in Lean. As such, I use the `half_pos` theorem with the `refine` tactic to change the goal to what is currently shown in the infoview. The `refine` tactic is useful because it tries to apply the arguments it is given to the final goal and then changes the goal to whatever is needed to meet the hypotheses in the arguments. In this case, `half_pos` claims that if you have some $a > 0$, then $\frac{a}{2} > 0$. The `refine` tactic then applies the result of that theorem and leaves us to show that $a > 0$, and Lean is smart enough to

figure out that we actually need to show $a - b > 0$.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2
  have h3 : ε > 0 := by
    refine half_pos ?h


      done


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: b < a
ε: ℝ := (a - b) / 2
⊢ 0 < a - b
```

Step 5

The `sub_pos` theorem says that $0 < a - b$ if and only if $b < a$, so we only need to apply this with our second hypothesis to complete our current goal. This finishes off the proof of the third hypothesis and allows us to return to the main theorem.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2
  have h3 : ε > 0 := by
    refine half_pos ?h
    exact Iff.mpr sub_pos h2
    done


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: b < a
ε: ℝ := (a - b) / 2
h3: ε > 0
⊢ False
```

Step 6

I now try to lay out the fourth and final hypothesis which will be used to find a contradiction with `h2`. This is another example of something being quickly explained in the paragraph style proof, but being more cumbersome to justify within Lean.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2
  have h3 : ε > 0 := by
    refine half_pos ?h
    exact Iff.mpr sub_pos h2
    done
  have h4 : a ≤ b + ε := by


    done


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: b < a
ε: ℝ := (a - b) / 2
h3: ε > 0
⊢ a ≤ b + ε
```

Step 7

I first apply `h1` which has a similar effect to using the refine tactic earler: it applies the result of an if-then statement and changes our goal to proving the if.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2
  have h3 : ε > 0 := by
    refine half_pos ?h
    exact Iff.mpr sub_pos h2
    done
  have h4 : a ≤ b + ε := by
    apply h1


    done


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: b < a
ε: ℝ := (a - b) / 2
h3: ε > 0
⊢ ε > 0
```

Step 8

Now that our goal has been properly modified, `h3` is the only other thing necessary to justify this hypothesis.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2
  have h3 : ε > 0 := by
    refine half_pos ?h
    exact Iff.mpr sub_pos h2
    done
  have h4 : a ≤ b + ε := by
    apply h1
    apply h3
    done


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: b < a
ε: ℝ := (a - b) / 2
h3: ε > 0
h4: a ≤ b + ε
⊢ False
```

Step 9

The `dsimp` tactic will do its best to automatically simplify anything it is given, in this case it substitutes our specific epsilon value in for the arbitrary epsilon. This will now allow us to use `h4` to find our contradiction.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2
  have h3 : ε > 0 := by
    refine half_pos ?h
    exact Iff.mpr sub_pos h2
    done
  have h4 : a ≤ b + ε := by
    apply h1
    apply h3
    done
  dsimp at h4


  done
```

Tactic State in Infoview

```
R: Type u_1
inst†: Ring R
ab: ℝ
h1: ∀ (ε : ℝ), ε > 0 →
  a ≤ b + ε
h2: b < a
ε: ℝ := (a - b) / 2
h3: ε > 0
h4: a ≤ b + (a - b) / 2
⊢ False
```

Step 10

The `linarith` tactic is quite powerful as it will attempt to simplify the goal as well as hypotheses and then look for a contradiction amongst the known hypotheses. This is one

example where Lean actually requires quite a bit less explanation than a typical proof. The majority of my paragraph style proof above was spent simplifying and manipulating `h4` and `h2`, whereas in Lean I need to specify none of that! It is quite impressive that Lean is already able to do so much simplification and even find contradictions with no user input. This ability will likely only increase in power in the future, and some developments have even occured during the planning and writing of this thesis that make other simplification tactics substantially more powerful.

Lean File

```
example (a b : ℝ) (h1 : ∀ ε : ℝ,
    ε > 0 → a ≤ b + ε) :
    a ≤ b := by
  by_contra h2
  push_neg at h2
  let ε := (a - b) / 2
  have h3 : ε > 0 := by
    refine half_pos ?h
    exact Iff.mpr sub_pos h2
    done
  have h4 : a ≤ b + ε := by
    apply h1
    apply h3
    done
  dsimp at h4
  linarith
  done
```

Tactic State in Infoview

```
No goals
```

Absolute Convergence

Paragraph Style Proof

Theorem. *Prove that* $\lim(x_n) = 0$ *if and only if* $\lim(|x_n|) = 0$.

Proof. ($\Longrightarrow$) First assume that $\lim(x_n) = 0$. Then for all $\varepsilon > 0$ we know there exists a $k_\varepsilon \in \mathbb{N}$ such that for all natural numbers $n > k_\varepsilon$, $|x_n - 0| < \varepsilon$. Thus $|x_n| < \varepsilon$ and also $||x_n| - 0| < \varepsilon$, so $\lim(|x_n|) = 0$.

($\Longleftarrow$) Now assume that $\lim(|x_n|) = 0$. Then for all $\varepsilon > 0$ we know there exists a $k_\varepsilon \in \mathbb{N}$ such that for all nautral numbers $n > k_\varepsilon$, $||x_n| - 0| < \varepsilon$. But $||x_n| - 0| = ||x_n|| = |x_n| = |x_n - 0|$. So $|x_n - 0| < \varepsilon$ and $\lim(x_n) = 0$. □

Lean Proof

Setting up the problem

Lean does not include a built in epsilon definition of a limit for sequences, so it is first necessary to define a limit in Lean. I use the following definition:

```
def ConvergesTo (s : ℕ → ℝ) (a : ℝ) :=
  ∀ ε > 0, ∃ N, ∀ n ≥ N, |s n - a| < ε
```

From this point we can set up our problem as normal.

Lean File

```
example (s1 : ℕ → ℝ) :

    ConvergesTo s1 (0 : ℝ) ↔

    ConvergesTo (abs s1) (0 : ℝ)

    := by


  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ

⊢ ConvergesTo s1 0

  ↔ ConvergesTo |s1| 0
```

Step 1

The first thing I ask Lean to do is rewrite the definition of convergence that I defined earler when it is used in our goal. This will allow us to actually use and work towards the information in both instances of `ConvergesTo` in the problem. The fully expanded definition is shown in the infoview panel.

Lean File

```
example (s1 : ℕ → ℝ) :

    ConvergesTo s1 (0 : ℝ) ↔

    ConvergesTo (abs s1) (0 : ℝ)

    := by

  rw [ConvergesTo]

  rw [ConvergesTo]



  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ

⊢ (∀ (ε : ℝ), ε > 0 →

    ∃ N, ∀ (n : ℕ),

    n ≥ N →

    |s1 n - 0| < ε) ↔

    ∀ (ε : ℝ), ε > 0 →

    ∃ N, ∀ (n : ℕ),

    n ≥ N →

    |abs s1 n - 0| < ε
```

Step 2

Here I set up a hypothesis which will later be used to rewrite both sides of the if and only if statement into something that is equal to the other.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by


    done


  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
x: ℕ
⊢ |s1 x| = |abs s1 x|
```

Step 3

In this instance Lean essentially already knows that our goal is true, and only need to be told to simplify using the definition of absolute value in order to verify this. While it is impressive that Lean requires little guidance, seeing some of the other things Lean is capable of leaves me a bit underwhelmed that Lean requires any input here. Because Lean is constantly being developed there may come a time where simple statements like this are automatically verified without any user input.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done

  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
h3: ∀ (x : ℕ), |s1 x| =
  |abs s1 x|
⊢ (∀ (ε : ℝ), ε > 0 →
    ∃ N, ∀ (n : ℕ),
    n ≥ N →
    |s1 n - 0| < ε) ↔
    ∀ (ε : ℝ), ε > 0 →
    ∃ N, ∀ (n : ℕ),
    n ≥ N →
    |abs s1 n - 0| < ε
```

Step 4

Here the `Iff.intro` tactic splits up the if and only if statement in the goal and allows us to prove each direction individually, as is often done in a paragraph style proof.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done
  apply Iff.intro
  · --Forwards


  · --Reverse


  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
h3: ∀ (x : ℕ), |s1 x| =
  |abs s1 x|
⊢ (∀ (ε : ℝ), ε > 0 →
    ∃ N, ∀ (n : ℕ),
    n ≥ N →
    |s1 n - 0| < ε) →
    ∀ (ε : ℝ), ε > 0 →
    ∃ N, ∀ (n : ℕ),
    n ≥ N →
    |abs s1 n - 0| < ε
```

Step 5

The `intro` tactic applied here allows me to assume the if part of an if-then statement and automatically names it with the hypothesis name I give it.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done
  apply Iff.intro
  · --Forwards
    intro h1


  · --Reverse


  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
h3: ∀ (x : ℕ), |s1 x| =
  |abs s1 x|
h1: ∀ (ε : ℝ), ε > 0 →
  ∃ N, ∀ (n : ℕ),
  n ≥ N → |s1 n - 0| < ε
⊢ ∀ (ε : ℝ), ε > 0 →
    ∃ N, ∀ (n : ℕ),
    n ≥ N →
    |abs s1 n - 0| < ε
```

Step 6

With the `simp` tactic, Lean attempts to simplify the current goal. In this case, the $||s1_n| - 0|$ is simplified to $||s1_n||$. This is now where our `h3` hypothesis can be applied, but

I will first attempt to simplify `h1`.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done
  apply Iff.intro
  · --Forwards
    intro h1
    simp


  · --Reverse


  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
h3: ∀ (x : ℕ), |s1 x| =
  |abs s1 x|
h1: ∀ (ε : ℝ), ε > 0 →
  ∃ N, ∀ (n : ℕ),
  n ≥ N → |s1 n - 0| < ε
⊢ ∀ (ε : ℝ), 0 < ε →
    ∃ N, ∀ (n : ℕ),
    N ≤ n →
    |abs s1 n| < ε
```

Step 7

The `simp` tactic has the same effect as in the previous step, but this time it is working on `h1` rather than the end goal. By default `simp` will attempt to work on the goal but if asked to it will attempt to simplify hypotheses as well.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done
  apply Iff.intro
  · --Forwards
    intro h1
    simp
    simp at h1


  · --Reverse


  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
h3: ∀ (x : ℕ), |s1 x| =
  |abs s1 x|
h1: ∀ (ε : ℝ), 0 < ε →
  ∃ N, ∀ (n : ℕ),
  N ≤ n → |s1 n| < ε
⊢ ∀ (ε : ℝ), 0 < ε →
    ∃ N, ∀ (n : ℕ),
    N ≤ n →
    |abs s1 n| < ε
```

Step 8

We can now use the reverse direction of `h3` to simplify our goal further. Notice that the leftwards facing arrow is necessary, as Lean typically tries to apply equalities from left to right. This means if the left side of the equality does not match what Lean is attempting to replace, Lean will not be able to rewrite in other terms.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done
  apply Iff.intro
  · --Forwards
    intro h1
    simp
    simp at h1
    simp [← h3]

  · --Reverse

  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
h3: ∀ (x : ℕ), |s1 x| =
  |abs s1 x|
h1: ∀ (ε : ℝ), 0 < ε →
  ∃ N, ∀ (n : ℕ),
  N ≤ n → |s1 n| < ε
⊢ ∀ (ε : ℝ), 0 < ε →
    ∃ N, ∀ (n : ℕ),
    N ≤ n →
    |s1 n| < ε
```

Step 9

The simplification done over the last few steps has modified both `h1` and our goal to be the same thing. Since we are assuming `h1` to be true, this allows us to apply that hypothesis and complete the first direction of our goal. Upon completion of the first goal, Lean automatically begins displaying the second goal, which can be solved quite similarly to the first. This is one place where some mathematicians may leave the other direction out of their paragraph style proof because it is essentially proved the same way as the previous direction. While this is easy and convenient, it never hurts to give a full explanation like what Lean will require of us.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done
  apply Iff.intro
  · --Forwards
    intro h1
    simp
    simp at h1
    simp [← h3]
    apply h1
  · --Reverse


  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
h3: ∀ (x : ℕ), |s1 x| =
  |abs s1 x|
⊢ (∀ (ε : ℝ), ε > 0 →
    ∃ N, ∀ (n : ℕ),
    n ≥ N →
    |abs s1 n - 0| < ε) →
    ∀ (ε : ℝ), ε > 0 →
    ∃ N, ∀ (n : ℕ),
    n ≥ N →
    |s1 n - 0| < ε
```

Step 10

With this direction I try to simplify in the same ways as before, but instead of using the leftwards direction of the equality in `h3`, I use the rightwards direction. This means that arrow does not need to be included and once again he have `h1` equal to our current goal.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done
  apply Iff.intro
  · --Forwards
    intro h1
    simp
    simp at h1
    simp [← h3]
    apply h1
  · --Reverse
    intro h1
    simp
    simp at h1
    simp_rw [h3]


  done
```

Tactic State in Infoview

```
s1: ℕ → ℝ
h3: ∀ (x : ℕ), |s1 x| =
  |abs s1 x|
h1: ∀ (ε : ℝ), 0 < ε →
  ∃ N, ∀ (n : ℕ),
  N ≤ n → |abs s1 n| < ε
⊢ ∀ (ε : ℝ), 0 < ε →
    ∃ N, ∀ (n : ℕ),
    N ≤ n →
    |abs s1 n| < ε
```

Step 11

With a hypothesis equal to our goal, we are able to apply the hypothesis and prove the other direction of the if and only if statement, completing the proof.

Lean File

```
example (s1 : ℕ → ℝ) :
    ConvergesTo s1 (0 : ℝ) ↔
    ConvergesTo (abs s1) (0 : ℝ)
    := by
  rw [ConvergesTo]
  rw [ConvergesTo]
  have h3 (x : ℕ) : |s1 x| =
      |abs s1 x| := by
    simp [abs]
    done
  apply Iff.intro
  · --Forwards
    intro h1
    simp
    simp at h1
    simp [← h3]
    apply h1
  · --Reverse
    intro h1
    simp
    simp at h1
    simp_rw [h3]
    apply h1
  done
```

Tactic State in Infoview

```
No goals
```

Convergence of a Specific Sequence

The following is an example of one situation where Lean is somewhat lacking in comparison to a paragraph style proof. The paragraph style proof is able to quickly and easily prove the desired end goal, but Lean has to work around a lot of the simple rewriting we would do in a normal proof. In this attempt to prove the convergence of a specific sequence, there were many issues with simplification involving arbitrary variables and the change from natural numbers to real numbers. These sorts of things can be easily explained in a paragraph style proof, but required significant work to prove in Lean.

I mentioned earlier that Lean's ability to simplify and make connections without user input was advancing quickly, and I encountered this when working on this problem. I originally had great difficulty getting Lean to accept that $2 = \frac{2(n+1)}{n+1}$, which is something which can easily be explained in a typical proof, but Lean had difficulty accepting. Thankfully, Lean has recently strengthened a tactic that renders much of my work here unnecessary. The `field_simp` tactic tries to simplify the current goal using what is known about all fields, and since we are working with the real numbers we are able to take advantage of this. I was not able to use this tactic since it was changed while I was working on the project, but seeing how quickly Lean is progressing is very promising.

Lean internally defines limits using filters and topology rather than the real analysis approach of epsilons, so the approach I was taking here is not the optimal approach for theorems involving limits in Lean. While this topological filter definition of a limit is very useful for the people who know how to use it, it makes Lean more difficult to use for those

who have not yet studied topology. Definitions such as this start to portray that Lean is not really something meant to be used for lower level mathematics, but rather complex and high level proofs.

Paragraph Style Proof

Theorem. *Prove that* $\lim(\frac{2n}{n+1}) = 2$.

Proof. Let $\varepsilon > 0$ and choose $k > \frac{1}{\varepsilon} - 1$ where $k \in \mathbb{N}$ by the Archimedean Property. Then for $n > k$:

$$\begin{aligned}\left|\frac{2n}{n+1} - 2\right| &= \left|\frac{2n}{n+1} - \frac{2(n+1)}{n+1}\right| \\ &= \left|\frac{-1}{n+1}\right| \\ &= \frac{1}{n+1} \\ &< \frac{1}{k+1} \\ &< \frac{1}{\frac{1}{\varepsilon} - 1 + 1} = \varepsilon.\end{aligned}$$

Thus we have that $\lim(\frac{2n}{n+1}) = 2$. □

Lean Proof

I will not attempt to walk through every step of this proof, but it suffices to show that Lean somewhat struggles when being used for lower-level proofs that it is not optimized for.

```
example : ConvergesTo (fun (n : ℕ) ↦
    ((2 * n) / (n + 1))) 2 := by
  intro ε
  intro h1
  obtain ⟨k, h13⟩ :=
    exists_nat_gt (2 / ε - 1) --Archimedean Property
  use k
  intro n
  intro h2
  dsimp
  have h3 : (2 : ℝ) = 2 * ((n + 1) / (n + 1)) := by
    have h4 : ((n + 1) / (n + 1)) =
        (n + 1) * ((n + 1) : ℝ)⁻¹ := by
      rfl
      done
    rw [h4]
    have h5 : (n + 1) * ((n + 1) : ℝ)⁻¹ = 1 := by
      rw [mul_inv_cancel]
      exact Nat.cast_add_one_ne_zero n
      done
    rw [h5]
    exact Eq.symm (mul_one 2)
```

```
  done
nth_rewrite 2 [h3]
have h6 : 2 * ((↑n + 1) : ℝ) / (↑n + 1) =
    ((2 * n) + 2) / (n + 1) := by
  rw [Distribute n]
  done
have h7 : 2 * (((↑n + 1) : ℝ) / (↑n + 1)) =
    2 * (↑n + 1) / (↑n + 1) := by
  rw [← mul_div_assoc 2 ((n + 1) : ℝ) ((n + 1) : ℝ)]
  done
rw [h7]
rw [h6]
rw [div_sub_div_same (2 * n : ℝ) (2 * n + 2) (n + 1)]
rw [sub_add_cancel']
rw [abs_div]
simp
have h8 : |(↑n + 1 : ℝ)| = ↑n + 1 := by
  simp
  apply LT.lt.le (Nat.cast_add_one_pos ↑n)
  done
rw [h8]
have h9 : (2 : ℝ) / (↑n + 1) ≤ 2 / (k + 1) := by
```

```
    apply div_le_div_of_le_left
    · --case 1
      linarith
      done
    · --case 2
      exact Nat.cast_add_one_pos k
      done
    · --case 3
      convert add_le_add_right h2 1
      apply Iff.intro
      · --subcase 1
        exact fun a => Nat.add_le_add_right h2 1
        done
      · --subcase 2
        intro h14
        apply add_le_add_right
        exact Iff.mpr Nat.cast_le h2
        done
      done
    done
  have h10 : 2 / (k + 1) < 2 / (2 / ε - 1 + 1) := by
    apply div_lt_div_of_lt_left
```

```
  · --case 1
    linarith
    done
  · --case 2
    simp
    apply div_pos
    linarith
    apply h1
    done
  · --case 3
    convert add_le_add_right h2 1
    apply Iff.intro
    · --subcase 1
      intro h11
      exact Nat.add_le_add_right h2 1
      done
    · --subcase 2
      intro h11
      have h12 : 2 / ε - 1 < (k : ℝ) := by
        simp only []
        apply h13
        done
```

```
      exact add_lt_add_right h12 1
      done
    done
  done
calc
  2 / (↑n + 1) ≤ (2 : ℝ) / (k + 1) := by
    apply h9
    done
  _ < (2 : ℝ) / (2 / ε - 1 + 1) := by
    apply h10
    done
  _ = ε := by
    ring_nf
    apply inv_inv
    done
done
```

3 Conclusions

The Good

It is easy to see the potential which Lean has to aid in proof verification, and it only seems to be gaining more popularity. The language is continuously being expanded and made more powerful which should only help make it accessible to more people. The language already has some features that could be very helpful to new users, such as the `apply?` tactic. This tactic looks at the current goal and all hypotheses and suggests tactics and theorems which could be applied to get closer to the goal. While this is certainly not perfect, it is very helpful and I used it quite a few times during my time with the language. This tactic is most useful for people who are new to the language and are not sure how to apply the theorems that they need or what the theorems may be called in Lean. This tactic worked best for me when completing simple steps that only required one more theorem already in Lean's library to reach my current goal.

Lean is like the kitchen we are cooking our proofs in and its libraries full of theorems are like cupboards and pantries containing all of the tools with which we are cooking. It would

be impossible for us to have read through the entire library and as such we can occasionally rely on the `apply?` tactic for some assistance when necessary. Picture a situation in which we already have boiled potatoes and we are attempting to make mashed potatoes, but we are unsure of which tool we could use to do this or in which cupboard it would be located. In this scenario, we could consult the `apply?` tactic as if we were asking, "What can I use to mash these potatoes up?" to which Lean may reply, "I would use the masher, found on the top shelf of that pantry over there." This tactic is not perfect and cannot make every connection, but if we are already quite close to a connection, `apply?` can typically get us the rest of the way there.

Another helpful tactic is the `sorry` tactic. This is mainly helpful for those trying to initially prove theorems in Lean rather than just convert previous proofs into Lean. Applying `sorry` to a goal automatically proves that goal within the context of the problem, allowing the user to continue on with the proof and return to that sub-proof later. Since using this tactic does not actually prove a theorem, Lean will underline the main theorem and not allow it to be used in any other proofs. While this line of thinking could also be done with pen and paper, the ability to see if one approach would even be fruitful before taking the time to justify everything is convenient and could help with the process of creating proofs.

`sorry` Example

This is an example of sorry being used in a `by_cases` proof.

```
example (a b c d : ℝ) (h1: a < b)
    (h2 : c ≤ d) : a + c < b + d := by
  by_cases h3 : c = d
  · --Case 1
    sorry
  · --Case 2
    push_neg at h3
    have h4 : c < d := by
      apply Ne.lt_of_le h3 h2
    exact calc
      a + c < b + c := add_lt_add_right h1 c
      _ < b + d := add_lt_add_left h4 b
  done
```

In this example, I may think that the first case where $c = d$ is simple and decide that I would rather focus my efforts on the second case first. Using `sorry` here allows me to only focus on the second case of the proof and still see if my justification for it would be valid. This tactic is not really about modifying the proof to make it easier to solve, but rather about a way of organizing our proof. This tactic also does not fit well into our cooking metaphor, because there is no way to skip over some steps and still get a finished product. This tactic may be more akin to a magical elf which would automatically make the sauce for our pasta with the stipulation that we can't actually eat any of it and we have to leave a note saying that the elf helped us.

Lean clearly laying out all known hypotheses and the current goal which would finish the proof could aid some users in figuring out how to prove theorems or problems they might have struggled with otherwise. This was one of the biggest benefits of the language when I was working with it, because it is often difficult to keep track of exactly what I know to be true. Being able to automatically simplify hypotheses and take things back to their definitions can greatly help figure out how each hypothesis could be applied.

Even without these tactics which make the language more user friendly, Lean still absolutely accomplishes its goal of automated proof verification. The language is somewhat difficult to get the hang of even with newly empowered tactics and theorems, but the users who have a deep understanding of the language are truly able to harness its full potential and easily work with difficult concepts in a way that is at times more concise than writing it out on paper. If a proof is being verified by another human, it may be necessary to go into great detail so that they are able to understand each step being taken. With Lean, however, once one knows how to work within Lean's realm of understanding they should be able to prove anything and not need to go into great detail. This difference in explanation would only grow larger when venturing into higher level mathematics and newly developing topics.

Lean's intent on proof verification also raises the question of how necessary automated proof verification truly is. No doubt is it useful to have a proof checked by a machine for you, but if this were to be used to develop new theorems, they would still need to eventually be written in a way on paper that other human mathematicians can easily understand and use. Automation in mathematics is nice, but ultimately the human

mathematicians are the ones making new developments and as such should be prioritized when writing proofs. Lean could, however, help with the writing of human readable proofs so long as the writer fully understands what the Lean code is doing. Lean can ensure that no logical jumps are being taken and if something is broken down into steps that Lean can understand, someone could translate them into human readable steps that would be totally coherent and logical.

The Bad

The biggest downside to using Lean is getting used to the programming language itself and trying to put mathematical steps into terms which Lean is able to understand. If someone has not spent much time working with programming languages it would certainly be difficult and time consuming to learn something entirely new, but even for those who have had experience programming the switch into Lean could still prove difficult. While there are methods within Lean which could aid new users such as the aforementioned `apply?` tactic, it is still necessary to know how to manipulate hypotheses into a form that Lean is able to work with, which may not always be as easy as it would be on paper.

Another downside to Lean is the lack of options and potentially the inability to immediately keep up with mathematics in the future. Development takes time, and in order to do proofs in Lean more theorems and definitions need to be added and worked out. As new areas of mathematics are expanded, it will be necessary to do upkeep on the language and add in new features or even make large overhauls to the language depending

on the significance of the work being done. As I mentioned earlier, Lean completely abandons some methods of proof in favor of more currently concise methods. This works fine for now, but if there was a newly developing area of mathematics which functioned better under a previously abandoned method, Lean may need to be reworked and theorems may need to be rewritten to fit better into the new system.

Lean also abandons some parts of mathematics that are not neccesarily applicable outside of exercises. While working on a problem in Lean I reached out on the Zulip forum for help and one person who contributes to the programming of Lean responded that what I was looking for wasn't included in Lean's library because it was really only useful for exercises. Cutting things like this may be necessary to save time programming the language, but it is a shame that everything cannot be included. Some simpler aspects of mathematics are important steps in developing ones understanding of higher level topics and to expect anyone coming into Lean to already know high level mathematics seems nearsighted. If Lean is to become more widely accepted, those learning mathematics need to be able to learn and use it as they grow in mathematical ability. This could help lead to mathematicians who are able to take full advantage of the language and even potentially contribute to development of the language.

The Future

With Lean constantly expanding and growing more powerful, it could become easy enough to use that it becomes widely used. The scene of users is quite active right now, and its

libraries are constantly being updated. Lean has even gained enough popularity that a few textbooks have been written about proving mathematical theorems using Lean. These vary in level of difficulty and some add their own libraries of theorems and definitions which are not included in the base installation of Lean. The *How To Prove It With Lean* textbook as well as *Mathematics in Lean* were both quite instrumental in learning the language and gaining a full grasp of Lean's capabilities. *How To Prove It With Lean* makes some substantial changes to the base language which I believe makes it easier to use than the base language. Due to the added tactics and simpler syntax rules in this book, I would certainly recommend any new users start with this book. Once the basics have been understood, *Mathematics in Lean* goes back to the base language and takes the language into higher level mathematics such as topology and calculus. These books do a great job of teaching the language and some of the math at the same time, making it possible for those starting to learn mathematical proofs to start in Lean and learn to take full advantage of it. If Lean continues to be supported and expanded we could certainly see a greater acceptance and implementation into mainstream mathematics.

Works Cited

This work had been formatted and styled from the book *How To Prove It With Lean*, written by Daniel J. Velleman. *How To Prove It With Lean* contains short excerpts from *How To Prove It: A Structured Approach, 3rd Edition*, by Daniel J. Velleman and published by Cambridge University Press.

Avigad, Jeremy, and Patrick Massot. 2020. "Mathematics in Lean." *Mathematics in Lean - Mathematics in Lean 0.1 Documentation.* https://leanprover-community.github.io/mathematics_in_lean/index.html#.

Avigad, Jeremy, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. 2021. "Theorem Proving in Lean 4." *Theorem Proving in Lean 4 - Theorem Proving in Lean 4.* https://lean-lang.org/theorem_proving_in_lean4/title_page.html.

Christiansen, David Thrane. 2023. "Functional Programming in Lean." *Functional Programming in Lean - Functional Programming in Lean.* https://lean-lang.org/functional_programming_in_lean/title.html.

Velleman, Daniel J. 2023. *How To Prove It With Lean*.

https://djvelleman.github.io/HTPIwL/.