Honors Trackwork—MATH 3125

Exploring Foundations with Lean

Ethan Barry

`ebarry2@patriots.uttyler.edu`

UTTyler™
THE UNIVERSITY OF TEXAS AT TYLER

1 Introduction

The fields of computer science and mathematics are very closely intertwined, and results or tools from one discipline often wind up supporting results in the other. An active area of research in programming language theory has been the development of "proof checkers," "proof assistants," or "theorem provers." These are programming languages which allow precise statements of mathematical theorems and their proofs, which enables a computer to deterministically check that the proof is correct.

One of the most widespread theorem provers available today is Lean,[1] which has been adopted not only by Fields Medalists Terence Tao[2] and Peter Scholze,[3] but by the broader mathematics community. It has also seen adoption by large companies such as Microsoft and Amazon, who use Lean to prove desirable properties of their systems always hold true.[4,5] In fact, one of Lean's principle developers, Leonardo de Moura, is a veteran of Microsoft Research and currently works at Amazon Web Services.

This paper will demonstrate how one uses Lean to formalize some proofs from the textbook while explaining the proofs themselves and how the Lean translations work under the hood. Simple proofs are a good way to gain hands-on experience with a technology that is certainly important in the software verification space, and may eventually be broadly useful in higher mathematics.

2 What is Lean?

We said Lean proves theorems, or checks proofs, but what does it mean for a computer to check a proof? Is it testing all possible values of the variables to check a proposition holds, or is there a better way? How can mathematicians or software engineers trust a computer program to prove something complex, and what advantages does a machine-checked proof offer?

2.1 Teaching Computers to Prove Things

The answer to the first question comes from programming language theory. In an older programming language like Pascal or C all variables are of some fixed *datatype*. The idea of a datatype was originally invented to specify a layout in physical memory; e.g. a `uint16_t` represents an unsigned integer of 16 bits. A datatype is somewhat like the mathematical concept of a set, because any particular `uint16_t` can take on an integer value from 0 to $2^{16} - 1$. Importantly, a variable's datatype must be known before the code is

1. Leonardo de Moura and Sebastian Ullrich, "The Lean 4 Theorem Prover and Programming Language," in *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings* (Berlin, Heidelberg: Springer-Verlag, 2021), 625–635, ISBN: 978-3-030-79875-8, https://doi.org/10.1007/978-3-030-79876-5_37, https://doi.org/10.1007/978-3-030-79876-5_37.

2. Terence Tao, *A Lean companion to "Analysis I"*, terrytao.wordpress.com, May 31, 2025, accessed December 6, 2025, https://terrytao.wordpress.com/2025/05/31/a-lean-companion-to-analysis-i/.

3. Kevin Hartnett, *Proof Assistant Makes Jump to Big-League Math*, Quanta Magazine, June 28, 2021, accessed December 6, 2025, https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/.

4. *Rewriting SymCrypt in Rust to modernize Microsoft's cryptographic library*, Microsoft, Inc., accessed December 8, 2025, https://www.microsoft.com/en-us/research/blog/rewriting-symcrypt-in-rust-to-modernize-microsofts-cryptographic-library/.

5. *Lean Into Verified Software Development*, Amazon Web Services, Inc., accessed December 7, 2025, https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/.

ever run. The compiler determines datatypes from the context, and makes sure they match the expected types in a process called *typechecking.*

Pascal and C required programmers to label every variable with a type, but algorithms were developed which allowed programmers to omit type labels and have the compiler infer them instead. The earliest example was Algorithm W, or the Hindley-Milner type system. These "type inference" algorithms gained power and became the modern theorem prover.

A concrete example would be the statement `1 + 1 = 2`. In Pascal this has datatype `boolean`, meaning it has a value from the set {true, false}. However in Lean, this has type `Prop`, which is very different. In some sense the Pascal `boolean` resolved to the value true. A `Prop` in Lean does not resolve to anything. The statement `1 + 1 = 2` is a value *all by itself.* If `boolean` is the set {true, false}, then Lean's `Prop` is the set of all logical propositions in Lean.

If Lean won't resolve `1 + 1 = 2` to be true or false, how do we know which it is? The answer is that we have to give Lean a proof of that fact. Lean's magic is that a proposition like `1 + 1 = 2` is *also* a type, and any proof of the proposition is a value of that type. Then a proof is valid if Lean's typechecker accepts it.

2.2 Why Bother?

The second question is more philosophical. The algorithm that infers and checks types in Lean is fairly straightforward, and relatively little code is dedicated to implementing it. So as long as users trust this typechecking algorithm and the code that implements it (the kernel), then they can trust any proof it accepts. If there's no bug in the kernel, Lean will only accept proofs that follow from the chosen axioms.

The advantages of Lean to mathematicians and software engineers come from the fact that Lean is code. For the last thirty years, developers have built tools to allow massive collaboration on codebases, such as Git. Now mathematicians are able to use these existing tools for collaboration on large-scale verification efforts, like the project to formalize Wiles' proof of Fermat's Last Theorem.[6]

3 First Steps & Basic Examples

Lean is much easier to understand through examples, and simple examples make for the best starting points. Typically proofs of theorems are written out in the normal way before they're ever translated to Lean. Sometimes even stating the theorem in a formal way (read "encoding it as a proposition in Lean") is difficult. So, for the first example we'll write out the theorem and prove it in English, and then present the formalization in Lean.

Because Lean can be used in many fields of mathematics where nice algebraic properties might not hold, it allows very careful control of steps in a proof. We can demonstrate this with a couple of extremely simple theorems that really need no proof in English. However, it's important that we be able to justify them to a computer.

Theorem 1. $1 + 1 = 2$.

6. Kevin Buzzard, *The Fermat's Last Theorem Project*, Lean Community Blog, April 30, 2024, accessed December 7, 2025, https://leanprover-community.github.io/blog/posts/FLT-announcement/.

Proof. $1 + 1$ becomes 2 by definition of addition on the integers, and $2 = 2$ is true because equality is reflexive. □

In Lean this looks like:

```
theorem one_plus_one_eq_two : 1 + 1 = 2 := by rfl
```

Breaking it down, the `theorem` keyword tells Lean we are stating a proposition. Next it expects a name for the theorem (we chose `one_plus_one_eq_two` for no particular reason). After the colon is the actual statement of the theorem. Then there's the := and the right-hand side, `by rfl`, which means "reflexive." The tactic `rfl` just proves anything of the form $a = a$.

Instructions like `rfl` are called "tactics," *and we'll make use of more tactics in other proofs.*

This might feel familiar to users of typed languages like Pascal. For example, compare the Lean code above with this Pascal code:

```
var counter : int := 5;
```

In the line of Pascal, `var` tells Pascal to expect a variable name, `counter`, which is followed by a colon and type annotation `int`, then the assignment operator := and the particular value from the datatype `int` that we choose, `5`. The Lean code has exactly the same structure, but with `theorem` instead of `var` and `1 + 1 = 2` where the type is supposed to be. But recall that in Lean, propositions are types themselves, so this is actually a type annotation! Then the right-hand side of the assignment operator := is one particular value from the datatype, or one particular proof of `1 + 1 = 2`. Lean does the addition automatically since it knows about the integers, and uses `rfl` to close the proof.

The first example is nice, but not very useful. This example will demonstrate using a variable in a proof of the following theorem. It also introduces a new tactic, `rw`, which means "rewrite," and uses a list of facts we know to rewrite the statement we're trying to prove.

Theorem 2. *If $x = 1$, then $x + 1 = 2$.*

Proof. The left-hand side is $1 + 1$ by substitution, and $1 + 1 = 2$ by the previous theorem. □

```
theorem x_plus_one_eq_two {x : ℤ} (hx : x = 1) : x + 1 = 2 := by rw [hx]; rfl
```

This theorem has two extra things on the left-hand side of the assignment. Both are hypotheses. The first, `x : ℤ`, says x is an integer, and the second is a hypothesis named `hx`, which says $x = 1$. On the right-hand side, `rw [hx]` tells Lean to rewrite all occurences of x as 1, since `hx` tells us that $x = 1$. Then we have the same goal as the last theorem, and we prove it with `rfl` again.

Theorem 3. *The product of odd numbers is odd.*

Proof. Let m and n be odd integers, that is $m = 2a + 1$ and $n = 2b + 1$ for some integers a and b. Then

$$\begin{aligned} mn &= (2a + 1)(2b + 1) \\ &= 4ab + 2a + 2b + 1 \\ &= 2(2ab + a + b) + 1. \end{aligned}$$

Let $k = 2ab + a + b$. We know $k \in \mathbb{Z}$ because the integers are closed under multiplication and addition, so mn has the form $2k + 1$, which is exactly the definition of an odd integer. Thus mn is odd. □

This is a straightforward proof, so we might hope it translates well to Lean. However, the first step is to state the theorem formally. We're saying that given two numbers of the forms $2a + 1$ and $2b + 1$, their product is $2k + 1$ where $k = 2ab + a + b$. Here's a direct translation of that idea:

```
theorem prod_odd_integers_is_odd {x y m n k : ℤ}
  (hx : x = 2 * m + 1) (hy : y = 2 * n + 1)
  (hk : k = 2 * m * n + m + n)
: x * y = 2 * k + 1 := by sorry
```

Right now this proof says `by sorry`*, which is a way of telling Lean that we aren't ready to prove it yet.*

Once again, notice how Lean is using the type system. The theorem's type is `x * y = 2 * k + 1`, so the assignment operator is expecting a value of that type, or a proof of that fact. Here the right-hand side is something different.

Let's walk through our first formalized proof for this theorem.

```
theorem prod_odd_integers_is_odd {x y m n k : ℤ}
  (hx : x = 2 * m + 1) (hy : y = 2 * n + 1)
  (hk : k = 2 * m * n + m + n)
: x * y = 2 * k + 1 := by
  calc
    x * y = (2 * m + 1) * (2 * n + 1) := by rw [hx, hy]
    _ = 2 * (2 * m * n + m + n) + 1 := by linarith
    _ = 2 * k + 1 := by rw [hk]
```

The theorem is the same, except now we have a proof body instead of `sorry`. This is called a `calc` block, and it's used to prove things involving transitive relations. This one walks through all the steps of the arithmetic in the English proof. Each line has a tactic for its justification. The new tactic `linarith` means high-school algebra. The "`_ =`" notation is just dragging the equality down the page.

It turns out `linarith` is very powerful. We can shorten this proof without using a `calc` block at all; `linarith` will simplify everything directly.

```
theorem prod_odd_integers_is_odd {x y m n k : ℤ}
  (hx : x = 2 * m + 1) (hy : y = 2 * n + 1)
  (hk : k = 2 * m * n + m + n)
: x * y = 2 * k + 1 := by rw [hx, hy]; linarith
```

Even though it's shorter, it's still a pretty disgusting way to write this. It would be much nicer to have a conclusion like `(x * y) odd` or something. Fortunately Lean comes with a definition of what it means to be even or odd, and we can use it directly in this next version of the proof.

```
theorem prod_odd_integers_is_odd2 {m n : ℤ}
  (hm : Odd m) (hn : Odd n)
  : Odd (m * n) := by
    rcases hm with ⟨a, ha⟩
    rcases hn with ⟨b, hb⟩

    rw [ha, hb]

    use 2 * a * b + a + b

    linarith
```

This uses Lean's proposition **Odd**, as well as the **rcases** tactic, which deconstructs a hypothesis into its components. For example, **hm** becomes a variable a and a hypothesis **ha** that $m = 2a + 1$. By applying **rcases** twice, and substituting these new variables and equations into the expression mn, we get that $mn = 2(2ab + a + b) + 1$. We use $2ab + a + b$ as our k in the oddness proposition that $mn = 2k + 1$. Finally we simplify by **linarith** and we're done.

4 Proofs by Contradiction

At this point we've met nearly all the tools we'll need for more complicated types of proof, such as proof by contradiction. The classic example of proof by contradiction is the proof that $\sqrt{2} \notin \mathbb{Q}$, so we'll try that next.

4.1 Proof that Square Root of Two is Irrational

There are a couple of points to keep in mind that will make the proof much easier to do. One is that we can prove any statement that's *equivalent* to $\sqrt{2} \notin \mathbb{Q}$, and we've proven the original statement. That means we can choose whichever form of the problem suits us best. Two is that Lean provides many theorems and definitions already, and we can invoke some of these to avoid reinventing the wheel. We'll avoid definitions that aren't in the course's textbook.

Now we must decide what form of the statement to prove. If $\sqrt{2} \notin \mathbb{Q}$, then there are no integers p and q such that $\sqrt{2} = \frac{p}{q}$. However, in this form, we have to worry about whether q is zero at multiple stages during the proof. This statement is algebraically equivalent to saying that $q\sqrt{2} = p$, and that seems nicer. Better still, we can square both sides to get $2q^2 = p^2$. Now we only have to deal with integers.

Recall that the crucial assumption in this proof is that p and q are relatively prime (coprime). This just means they have no common factors greater than 1. Lean provides a definition of "coprime," but it won't help us much. We can use Lean's definition of GCD to express this idea instead. We're ready to state the theorem in Lean. Notice the hypothesis we provide, that `p.gcd q = 1`, which is Lean's way of writing $\gcd(p, q) = 1$.

```
theorem root_two_irrational {p q : ℤ} (hmn : p.gcd q = 1) : p ^ 2 ≠ 2 * q ^ 2 := by
  sorry
```

Since this is a proof by contradiction, we have to tell Lean to expect a contradiction to occur. We do this with the line `by_contra h`, which says we're assuming the negation of the thing we want to prove and naming that hypothesis `h`. Then it will begin to look for things it knows which conflict. The algorithm that finds contradictions is pretty good, but it can't see the contradiction at this stage. We have to explain ourselves. We're going to use some `have` blocks to establish other facts we know.

There are two copies of the hypothesis that $2 \mid p$. We will "destroy" one of them later, so I declared a copy with a tick after the name.

We also need to invoke other facts Lean already knows. One is that if p is prime, and $p \mid a^n$, then $p \mid a$. This theorem has a name in Lean, which is `Prime.dvd_of_dvd_pow`.[7] It needs two hypotheses. One, that p is actually prime (Lean knows 2 is prime, and that theorem is `Int.prime_two`), and two, that $p \mid a^n$ for some integer a. That's why we need `hq_sq_even`, which says $2 \mid q^2$. Watch for that on line 4.

```
1  theorem root_two_irrational {p q : ℤ} (hmn : p.gcd q = 1) : p ^ 2 ≠ 2 * q ^ 2 := by
2    by_contra h
3    have hp_sq_even : 2 ∣ p ^ 2 := by use q ^ 2
4    have hp_even : 2 ∣ p := by exact Prime.dvd_of_dvd_pow Int.prime_two hp_sq_even
5    have hp_even' : 2 ∣ p := by exact hp_even
6    sorry
```

Now that we know p is even, we can explore what that implies. There must be some $k \in \mathbb{Z}$ such that $2k = p$, and we can tell Lean that with the `rcases` trick we saw at the end of the last section.

```
1  theorem root_two_irrational {p q : ℤ} (hmn : p.gcd q = 1) : p ^ 2 ≠ 2 * q ^ 2 := by
2    by_contra h
3    have hp_sq_even : 2 ∣ p ^ 2 := by use q ^ 2
4    have hp_even : 2 ∣ p := by exact Prime.dvd_of_dvd_pow Int.prime_two hp_sq_even
5    have hp_even' : 2 ∣ p := by exact hp_even
6
7    rcases hp_even' with ⟨k, hk⟩
8    · rw [hk, mul_pow] at h
9      dsimp at h; symm at h
10     have hq_sq : q ^ 2 = 2 * k ^ 2 := by linarith
11     have hq_sq_even : 2 ∣ q ^ 2 := by use k ^ 2
12     have hq_even : 2 ∣ q := by exact Prime.dvd_of_dvd_pow Int.prime_two hq_sq_even
13   sorry
```

The `rcases` trick is what "destroys" or "destructures" the copy of `hp_even`. It gives us a new variable k, and the hypothesis `hk`, or $p = 2k$. The two lines after that do substitution and transform our main hypothesis `h` into $4k^2 = 2q^2$. Lines 10, 11, and 12 allow us to conclude that $2 \mid q$ as well. From line 4, we know $2 \mid p$. Now we're ready to go for the contradiction. The last three lines of the proof are below.

These lines replace the `sorry` *on line 13 above.*

```
13   have h_dvd_gcd : 2 ∣ p.gcd q := by exact Int.dvd_gcd hp_even hq_even
14   rw [hmn] at h_dvd_gcd
15   contradiction
```

7. Documentation: leanprover-community.github.io/mathlib4_docs/

Line 13 invokes a theorem provided by Lean, `Int.dvd_gcd`. This says that if $c \mid a$ and $c \mid b$ then $c \mid \gcd(a, b)$. It takes as "arguments" the hypotheses that $2 \mid p$ and $2 \mid q$, and gives us the result that $2 \mid \gcd(p, q)$. Yet we know by hypothesis `hmn` that $\gcd(p, q)$ is 1, so we can rewrite that. Lean is now faced with the false statement we have just proved, that $2 \mid 1$, and we declare a contradiction.

4.2 Proof that an Irrational Product has an Irrational Factor

This section will demonstrate another proof by contradiction, as well as how to write our own definitions and use some more powerful proof tactics. Let's prove it in English first, and then translate that to Lean.

Theorem 4. *If a is rational and for some real number b, ab is not rational, then b is irrational.*

Proof. Suppose for the sake of contradiction that b is rational. Since a is rational, we can say $a = \frac{m}{n}$ for some $m, n \in \mathbb{Z}$ with $n \neq 0$, and since b is rational, we can say $b = \frac{r}{s}$ for some $r, s \in \mathbb{Z}$ with $s \neq 0$. Then the product of a and b can be expressed as

$$ab = \left(\frac{m}{n}\right)\frac{r}{s} = \frac{mn}{rs},$$

which, since by closure of the integers under multiplication mn and rs are integers with $rs \neq 0$, is a rational number. Yet we assumed at the outset that ab was irrational, thus we have a contradiction, and b must be irrational. □

To translate this theorem into Lean, we first need an idea of what it means to be a rational number. Lean provides definitions for things like these, but we can define our own easily. A number a is rational if there exist some integers m and n such that $a = \frac{m}{n}$. Then in Lean, what we want is a proposition that these integers exist, given some number a. Here's the translation:

Remember that the Lean type `Prop` *means any logical proposition. It's basically the set of all propositions.*

```
def IsRational (a : ℝ) : Prop := ∃ m n : ℤ, n ≠ 0 ∧ a = m / n
```

This is a proposition on some number a, that there exist $m, n \in \mathbb{Z}$ such that $n \neq 0$ and $a = \frac{m}{n}$, as we desired. Any irrational number is one for which the negation of this is true. Now, our theorem.

```
theorem irrat_prod_of_rat_imp_irrat {a b : ℝ}
  (ha_rat : IsRational a) (hab : ¬ IsRational (a * b))
: ¬ IsRational b := by
  by_contra h
  rcases ha_rat with ⟨m, n, hn_nonzero, ha⟩
  rcases h with ⟨r, s, hs_nonzero, hb⟩
  have hab_contra : IsRational (a * b) := by
    use (m * r); use (n * s)
    norm_num; split_ands
    exact hn_nonzero
    exact hs_nonzero
```

```
      rw [ha, hb]; field_simp
    contradiction
```

We'll examine only the new details here. The first line of the proof has the `by_contra h` tactic, just like last time. This just gives us an assumption `IsRational b`. Lines 7 and 8 "decompose" our assumptions into their component parts. For example, `ha_rat`, the theorem that a is rational, becomes two variables m and n, as well as two hypotheses, that $n \neq 0$ and that $a = \frac{m}{n}$.

Line 9 onwards is convincing Lean that ab is rational. We tell Lean that the numerator and denominator will be mr and ns, then use algebra to simplify expressions and invoke `split_ands`, which divides the proposition we're trying to prove (`IsRational`) into multiple subpropositions. Lines 12, 13, and 14 resolve each of these, and show that ab is rational. We know from `hab` that this is not the case, so we claim a contradiction.

Again, when we want to apply a theorem we already know, and something in the current context is in the form it expects, we use `exact`.

5 Proofs by Counterexample

Proof by counterexample is useful when proving or disproving statements with quantifiers. There are two types of proof by (counter)example. In the first, we're trying to show a universal quantifier is false. Then we need an example which fails the predicate. In the second, we're trying to show an existential quantifier is true, so we need an example which passes the predicate. Here's a demonstration of the first type.

Theorem 5. *Prove or disprove: For all real numbers x, $2^x \geq x + 1$.*

Proof. We will disprove the statement, that is, we'll prove its negation:

$$\neg\forall x \in \mathbb{R}, 2^x \geq x + 1 \implies \exists x \in \mathbb{R}, 2^x < x + 1.$$

We'll use the example $\frac{1}{2}$. We know $x^{\frac{1}{2}} = \sqrt{x}$, so we just want to show that $\sqrt{2} < \frac{3}{2}$. We also know that if $\sqrt{x} < y$ when $x, y \geq 0$, then $x < y^2$. (We could do the algebra to get this, or we could just invoke it.) So now all we have to show is that

- $2 \geq 0$,
- $\frac{3}{2} \geq 0$,
- and $\sqrt{2} < \frac{3}{2}$,

and we can apply that idea. Since all of these are true, we've found a counterexample to the original statement. □

That was certainly a less clean proof than we could have written, but it translates very well into Lean, which likes a fine-grained justification for each step. (Again, because it's built for areas of math where nice properties don't always work.) In fact, the proof written as above looks almost exactly like the Lean version.

```
theorem counterexampledemo : ¬ ∀ x : ℝ, 2 ^ x ≥ x + 1 := by
  simp
  use (1 / 2)
  rw [← Real.sqrt_eq_rpow, Real.sqrt_lt]
  · norm_num
  · norm_num
  · norm_num
```

The **simp** tactic passes the negation through the quantifier like we did in the English proof, then we suggest the example of $\frac{1}{2}$, and invoke two facts in one rewrite step. First is that $x^{\frac{1}{2}} = \sqrt{x}$, and second is Lean's version of the theorem we invoked. Then we just have to prove the three things in the list, which were all obvious. We can use **norm_num** for obvious things, and that completes the proof.

Lean has an ecosystem of tools to find useful theorems in the mathematical "standard library," such as "Loogle," the Lean version of Haskell's old search tool "Hoogle." Yes, computer scientists are bad at naming things.

6 Proof by Contraposition

Contraposition is a technique which takes advantage of the fact that the proposition $P \implies Q$ is logically equivalent to $\neg Q \implies \neg P$. When one direction is very hard to prove, it's possible the other direction is simple. We can use contraposition on a hypothesis **h** with the Lean tactic **contrapose h**.

6.1 Proof that 5 doesn't divide factors if it doesn't divide their product

This was an example from one of the lectures, which would probably be hard to prove without contraposition. Proving a negative is difficult, but the contraposition of this theorem becomes simple. The English proof is below.

Theorem 6. *If* $5 \nmid ab$ *then* 5 *divides neither* a *nor* b.

Proof. The contraposition of this theorem is that if $5 \mid a$ or $5 \mid b$, then $5 \mid ab$. If we prove the contraposition, then we've proven the original theorem. We can do this by cases.

Case I. $5 \mid a$ By definition of "divides" there exists some k such that $5k = a$. Then $ab = 5kb$ and $5 \mid ab$ by definition of "divides."

Case II. $5 \mid b$ By definition of "divides" there exists some k such that $5k = b$. Then $ab = 5ak$ and $5 \mid ab$ by definition of "divides."

In either case, $5 \mid ab$, so we have proven the contraposition is true. □

In class, we used the magic words "without loss of generality" to sidestep the repetition in Case II. There are ways to do this in Lean (usually via the **wlog** tactic) but this is a simple example and **wlog** is complicated. We'll just prove both cases directly in Lean.

```
theorem contrapositiondemo {a b : ℤ}
  (hab : ¬ 5 ∣ a * b)
: ¬ (5 ∣ a) ∧ ¬ (5 ∣ b) := by
  split_ands
  · contrapose hab
    simp_all
```

```
9      obtain ⟨k, hk⟩ := hab
10     dsimp [(· ∣ ·)]
11     rw [hk]
12     use k * b; ring
13   · contrapose hab
14     simp_all
15     obtain ⟨k, hk⟩ := hab
16     dsimp [(· ∣ ·)]
17     rw [hk]
18     use k * a; ring
```

We start the proof with `split_ands` to break up the logical AND in the theorem. To Lean, this gives us two cases to prove. We still haven't done any contraposition yet; this is just a technical step. In the two blocks below that (lines 7 & 13) we do almost the same thing twice, just like in the English proof. We call `contrapose hab` to get the contraposition of our hypothesis and goal, and then `simp_all` to eliminate the double negatives Lean gave us. This proof also uses a new tactic, `obtain`, which works like `have` and gives us some fact we know in the context of the proof.

Now `hab` tells us that $5 \mid a$ (or $5 \mid b$ in the other case). We destructure it using `obtain` on line 9 (15) into a new variable k and the hypothesis `hk`, which says $a = 5k$ ($b = 5k$). Then we use `dsimp` to destructure the "divides" relation found in our goal. (The dots are "placeholders" for the arguments.) Then our goal becomes $\exists c \in \mathbb{Z}, ab = 5c$, and we provide an example with `use k * b` (or `use k * a` in the other case). The `ring` tactic just confirms the result will be an integer, since the integers are closed under multiplication.

7 Proofs by Induction

Lean makes induction proofs fairly easy. This isn't very surprising; most "functional" languages like Lean use recursion and induction very heavily under the hood. In Lean, the natural numbers themselves are defined recursively. We'll look at one of the homework problems as an example.

This is the first example using the natural numbers in Lean. Because Lean was written by computer scientists, it uses the (correct!) convention that $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, *as you can see in the proof.*

Theorem 7. *If n is a natural number, then* $3 \mid n^3 + 5n + 6$.

Proof. By induction on n. If $n = 0$, it is true because $3 \mid 6$. For the inductive step, if $3 \mid n^3 + 5n + 6$ for some n, then for some natural number k, $n^3 + 5n + 6 = 3k$ and

$$\begin{aligned}(n+1)^3 + 5(n+1) + 6 &= n^3 + 3n^2 + 3n + 1 + 5n + 5 + 6 \\ &= (n^3 + 5n + 6) + 3n^2 + 3n + 6 \\ &= 3k + 3n^2 + 3n + 6 \\ &= 3(k + n^2 + n + 2),\end{aligned}$$

which means $3 \mid (n+1)^3 + 5(n+1) + 6$ by definition of "divides." So it holds for any natural number $n + 1$ as long as it holds for n, and we have proved it for all natural numbers by induction. □

To pull off an induction proof in Lean, we use the `induction` tactic. After we specify which variable we're doing induction on, this tactic needs us to prove two cases, "zero" and "succ n," with "succ" being the successor function. Here's the code.

```
1  theorem inductiondemo {n : ℕ} : 3 ∣ n ^ 3 + 5 * n + 6 := by
2    induction n with
3    | zero => norm_num
4    | succ n ih =>
5        obtain ⟨k, hk⟩ := ih
6        dsimp [(· ∣ ·)]
7        let c := k + n ^ 2 + n + 2
8        have hn : (n + 1) ^ 3 + 5 * (n + 1) + 6 = 3 * c := by
9          calc
10         (n + 1) ^ 3 + 5 * (n + 1) + 6 = 3 * k + 3 * n ^ 2 + 3 * n + 6 := by linarith
11         _ = 3 * (k + n ^ 2 + n + 2) := by linarith
12       use c
```

In this proof, the zero case is easy and only takes one line. Remember, `norm_num` handles easy arithmetic, and the zero case just needs to show that three divides six. The next case on line four says `succ n`, or $n+1$, and `ih`, which is short for "inductive hypothesis." It's just the local assumption that the proposition holds for n.

Line 5 destructures the inductive hypothesis into the assumption that $n^3 + 5n + 6 = 3k$ for some natural number k. Then we use `dsimp` to destructure "divides" again. We name a variable c and set it equal to $k + n^2 + n + 2$, which we use in proving that $(n+1)^3 + 5n + 5 + 6 = 3c$, which by definition of "divides," tells us that the proposition holds for $n + 1$ when it holds for n. The `calc` block jumps through all our work in the English proof with two lines and the `linarith` tactic, which just solves linear equations and applies basic algebra.

8 Conclusion

Here we've covered most of the proof techniques discussed in the class, with at least one formalized example for each. They demonstrate what different kinds of proofs in Lean look like, as well as some of the more powerful tools available. Working on this report also highlighted some of the difficulties of working in Lean. Searching the Internet for help resources generally won't be useful. Search engines like Google aren't optimized for math and there isn't enough data yet for a search to be helpful. There are message boards where community members (usually professional mathematicians) are active and respond to questions, and this appears to be the best way to seek help.[8]

It will be interesting to see where Lean goes in the coming years, especially in the safety-critical software space. Verifying that a program can always recover/never halts or faults is very important in aerospace,

8. The majority of the discussion happens here: https://leanprover.zulipchat.com/

automotive, and medical applications. It also appears to have some potential in mathematics, with ongoing efforts to formalize large and cumbersome proofs that take a long time to check by hand.

BIBLIOGRAPHY

Buzzard, Kevin. *The Fermat's Last Theorem Project*. Lean Community Blog, April 30, 2024. Accessed December 7, 2025. https://leanprover-community.github.io/blog/posts/FLT-announcement/.

Hartnett, Kevin. *Proof Assistant Makes Jump to Big-League Math*. Quanta Magazine, June 28, 2021. Accessed December 6, 2025. https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/.

Lean Into Verified Software Development. Amazon Web Services, Inc. Accessed December 7, 2025. https://aws.amazon.com/blogs/opensource/lean-into-verified-software-development/.

Moura, Leonardo de, and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language." In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, 625–635. Berlin, Heidelberg: Springer-Verlag, 2021. ISBN: 978-3-030-79875-8. https://doi.org/10.1007/978-3-030-79876-5_37. https://doi.org/10.1007/978-3-030-79876-5_37.

Rewriting SymCrypt in Rust to modernize Microsoft's cryptographic library. Microsoft, Inc. Accessed December 8, 2025. https://www.microsoft.com/en-us/research/blog/rewriting-symcrypt-in-rust-to-modernize-microsofts-cryptographic-library/.

Tao, Terence. *A Lean companion to "Analysis I"*. terrytao.wordpress.com, May 31, 2025. Accessed December 6, 2025. https://terrytao.wordpress.com/2025/05/31/a-lean-companion-to-analysis-i/.