Learning Logic and Proof with an Interactive Theorem Prover

Jeremy Avigad

April 6, 2018

Abstract

A course developed by Robert Y. Lewis, Floris van Doorn, and the author serves as an undergraduate introduction to mathematical proof, symbolic logic, and interactive theorem proving. The treatment of each topic on its own is routine, and the novelty lies in the way they are combined to form a multifaceted introduction to mathematical reasoning and argumentation. Students are required to master three different languages: informal mathematical language, formal symbolic logic, and a computational proof language that lies somewhere in between. Experience teaching the course suggests that students have no trouble keeping the languages distinct while at the same appreciating the relationships between them, and that the multiple representations support one another and provide a robust understanding of mathematical proof.

1 Introduction

Although mathematical language and the canons of proof have been evolving for centuries, the fundamentals have remained remarkably stable over time. High school students can still profitably read Euclid's *Elements* and follow the deductive structure of the arguments. They can similarly make sense of Cardano's solution to the cubic equation and read Euler's proof of the theorem that if p is a prime number and a is not divisible by p, then a^{p-1} is congruent to 1 modulo p.

In contemporary terms, the definition of a prime number might be rendered as follows:

Definition 1.1. *A natural number p is prime if and only if $p \geq 2$ and for every n, if $n \mid p$, then $n = 1$ or $n = p$.*

Symbolic logic provides idealized languages to model informal mathematical vernacular, and in a language in which variables are assumed to range over the natural numbers, the definition of a prime number might be rendered as follows:

$$\mathit{Prime}(p) \equiv p \geq 2 \wedge \forall n\ (n \mid p \to n = 1 \vee n = p).$$

We logicians like to think that symbolic logic helps students understand mathematical language and proof, since it is specifically designed to codify the rules and norms that govern mathematical reasoning. This article provides at least anecdotal evidence that this is the case, describing a course that teaches symbolic logic and ordinary mathematical proof together. When we teach this course, we ask the students to recognize that we are dealing with two different languages and keep them

distinct, donning a mathematician's hat and a logician's hat at different times. This provides complementary perspectives on mathematical reasoning and encourages students to think about the relationships between the two.

A novel feature of the course is that students also learn to use a third language, namely, the language of the *Lean* interactive theorem prover [3]. In this system, the definition of a prime number can be rendered as follows:

```
def prime (p : ℕ) := p ≥ 2 ∧ ∀ n, n ∣ p → n = 1 ∨ n = p
```

Such definitions are parsed and checked for grammatical correctness by the system. Students are asked to write proofs that can be checked by the system as well, in a proof language that looks much like a programming language. This requires them to don yet a third hat, namely, that of the computer scientist, and put up with the trials and tribulations of formal specification.

The goal of the course is to teach students to read, write, and understand ordinary mathematical proofs. There are independent reasons to teach students symbolic logic, which an interesting branch of mathematics in its own right, with applications to philosophy, computer science, linguistics, and other disciplines. There are also independent reasons to teach students how to use an interactive theorem prover: not only are such systems becoming important tools for the verification of complex hardware and software systems, but, more generally, facility with logical formalisms and specification languages is important in a number of branches of computer science, ranging from programming languages and database theory to artificial intelligence. But these are not explicitly addressed in the course, and we take these benefits to be ancillary. The course remains focused on good old-fashioned mathematics.

Robert Y. Lewis, Floris van Doorn, and I have taught the course to undergraduates at Carnegie Mellon over the past three years, based on course materials that we are still developing [1]. These are freely available online:

http://leanprover.github.io/logic_and_proof

The textbook can be read in a browser, and clicking on examples and exercises in Lean launches a version of the program that also runs in the browser. A PDF version of the textbook is also available for download, and Lean can be installed and run independently. Running Lean in the browser is slower than running a native version, but it is more than adequate for the purposes of the course.

2 Course outline

The course was taught with three weekly 50-minute meetings over the course of a 14-week semester. The introductory chapter proclaims to students that the goals are as follows:

- to teach you to write clear, "literate," mathematical proofs
- to introduce you to symbolic logic and the formal modeling of deductive proof
- to introduce you to interactive theorem proving
- to teach you to understand how to use logic as a precise language for making claims about systems of objects and the relationships between them, and specifying certain states of affairs

We tell students that they will be learning three separate languages and that they should take care not to conflate them. We also tell them that the course can be viewed simultaneously as an introduction to mathematical proof, and introduction to symbolic logic, and an introduction to interactive theorem proving. We emphasize, however, that the latter two components are carried out in service of the first.

The first third of the course focuses on symbolic logic. We begin with a logic puzzle by George Sumner, "Malice and Alice," that presents a list of assertions about a cast of characters consisting of Alice, her husband, her brother, her son, and her daughter, and the murder of one by another. Students are asked to work out a solution to the problem on their own, but then we turn to an analysis of the language and form of Sumner's published solution. We ask students to reflect on the role of words and phrases like "and," "or," "if ... then," and "this is impossible," the last of which signals a contradiction. We then introduce symbols corresponding to these verbal constructions, and begin to analyze the rules governing their use.

Our introduction to symbolic logic is fairly conventional. We first present propositional logic and introduce natural deduction as our system of formal proof. We describe truth-table semantics, and explain the notions of soundness and completeness informally. We then continue with the language of first-order logic, including function and relation symbols, quantifiers, and equality. Once again, we present the relevant proof rules in natural deduction, followed by an informal discussion of the semantics. We get students to recognize that a first-order sentence can be interpreted in various structures, and that the role of the proof system is to certify certain entailments as *valid*, which means that the conclusion holds in any structure that satisfies the hypotheses.

The main novelty is that the conventional presentation is accompanied by an introduction to the use of logic in Lean. As we go, we show students how to write formal expressions and proofs that can be checked automatically. This material is treated in separate chapters in the textbook, with a clearly delineated shift in the exposition. Roughly every third class, we set aside conventional lectures to project a laptop display to the front of the classroom and work through examples and proofs together. Weekly homework assignments reflect a similar balance: roughly one third of the exercises are carried out formally in Lean, with the remainder consisting of conventional pen-and-paper exercises.

The remaining two thirds of the course provide a similarly conventional introduction to the fundamental concepts of mathematics and mathematical proof. We present elementary set theory, the conventional operations on sets, and the usual set identities. We talk about relations, including order relations and equivalence relations. We deal with functions and their general properties. We then turn to the natural numbers and proof by induction. With these fundamentals in place, we illustrate their use with quick tours of key mathematical topics: elementary number theory, finite combinatorics and counting principles, a construction of the real numbers via Cauchy sequences, and the theory of the infinite. A final chapter brings the logical foundations and the informal mathematics together by presenting and exploring the axioms of set theory.

In this latter part of the course, symbolic logic plays a more limited role. Initially, we show students how informal proofs of set identities can be carried out in natural deduction, but while natural deduction is useful for modeling the building blocks of proof, once those building blocks have been established, the utility of the system diminishes quickly. Natural deduction does not scale to writing real mathematical arguments, and laboriously representing long mathematical arguments in those terms would produce no new insights. This is not to say that logic has no role in clarifying concepts: later in the course, we return to logic to cast the principle of induction in symbolic terms and observe that it quantifiers over predicates. Similarly, we ask students to rec-

ognize that the statement "a function is surjective if and only if it has a right inverse" involves an implicit quantification over functions. The set-theoretic axioms, at the very end of the course, are presented explicitly in the first-order language of set theory. This return to symbolic logic bringss the presentation full circle.

When it comes to modeling ordinary mathematical proof in formal terms, Lean allows us to get further than natural deduction. The chapters on sets, relation, functions, and the natural numbers are also tracked by chapters that show how to reason about them in Lean. Lean's logic is very expressive, so that, in principle, *any* ordinary mathematical theorem can be formalized and proved in the system. But the course is not a course in interactive theorem proving, and teaching students to use the range of Lean's features and libraries falls well outside the scope. Rather, we focus on small examples: set theoretic identities, elementary properties of relations and functions, and simple proofs by induction. As with logical reasoning, the point is primarily to solidify student understanding of the patterns of reasoning by showing them the formal analogues. Thus we provide students with just enough information to carry out basic, illustrative homework examples, and ask them to carry out more complicated pen-and-paper proofs in ordinary mathematical language.

Over time, we expect Lean to evolve to the point where we will be able to use automation to support writing more complicated mathematical proofs in a style that is close the informal ones we expect students to write. As a result, we ultimately hope to extend the textbook with Lean-based chapters on combinatorics, number theory, the real numbers, and so on. In the meanwhile, the approach we have adopted is decidedly pragmatic: we use Lean for as long as it is a useful tool for helping students to understand the logical structure of mathematical concepts and proofs, and otherwise rely on conventional mathematical presentations.

3 Formal perspectives on proof

In this section, I will discuss the ways that using an interactive theorem prover can help students understand the logical structure of mathematical statements and proofs. Representing natural language assertions in symbolic terms takes some getting used to. We explain to students that, in the language of propositional logic, we might assign propositional variables to basic assertions as follows:

- A: Alice's husband was in the bar
- B: Alice was on the beach
- C: Alice was in the bar
- D: Alice's husband was on the beach

With that assignment, the statement "either Alice's husband was in the bar and Alice was on the beach, or Alice was in the bar and Alice's husband was on the beach" is rendered as

$$(A \wedge B) \vee (C \wedge D).$$

In Lean, we can declare propositional variables and write the expression as follows:

```
variables A B C D : Prop
#check (A ∧ B) ∨ (C ∧ D)
```

The `#check` command provides immediate feedback to the student and confirms that the expression is a well formed proposition. Lean is designed as a research tool and not for pedagogical purposes, and error messages can be cryptic. But they are generally good enough for students to locate problems and fix them, with some trial and error. For that purpose, Lean's real-time feedback is essential.

For another example, after we introduce relations and quantifiers, students learn that they can express the fact that a binary relation $<$ is dense by writing

$$\forall x, y\ (x < y \to \exists z\ (x < y \wedge y < z)).$$

In Lean, they can declare a binary relation on a datatype, A, assign the $<$ notation, and express the assertion as follows:

```
variables (A : Type) (R : A → A → Prop)
local infix ` < ` := R

#check ∀ x y, x < y → ∃ z, x < z ∧ z < y
```

The correspondence between the usual notation of first-order logic and Lean's syntax is not exact. For example, in first-order logic one usually fixes the underlying domain of all the variables and functions, whereas in Lean one declares the underlying data type `A` explicitly and models the relation `R` as a function which takes two elements of `A` and returns a proposition. (In computer science and interactive theorem proving, it is common to take the binary arrow operation in `R : A → A → Prop` to associate to the right, to make declarations like this convenient.) Another difference is that in mathematical logic it is common to take quantifiers to bind tightly, which means that the parentheses in the first rendering above are needed to make the scope of the universal quantifier over x and y bind the rest of the formula. In contrast, in computer science, it is common to give quantifiers the widest scope possible, which explains why no parentheses are needed in Lean's formulation. We do not shy away from explaining this to students: learning to use a symbolic formal language requires learning the relevant conventions. To avoid overwhelming them, we do our best to limit the information they need to understand the examples and carry out the homework assignments. To our pleasant surprise, we have found that minor syntactic differences between the our three languages—informal mathematical language, symbolic logic, and Lean—do not cause problems. Students are capable of moving between the different linguistic contexts while at the same time recognizing them as alternate representations of a common logical structure.

In the course, we use both natural deduction and Lean to convey to students the common patterns and idioms of mathematical proof. For example, informally, we tell students that in order to prove a statement of the form "if A then B," they should assume A hypothetically and use that assumption to argue that B holds. As a natural deduction proof rule, this pattern is represented as follows:

$$\dfrac{\begin{array}{c}\overline{A}\\ \vdots\\ B\end{array}}{A \to B}$$

Here the line over the hypothesis A indicates that the hypothesis is *canceled* when the inference is complete. The vertical ellipsis represents the argument for B. In Lean, the pattern is rendered as follows:

```
variables A B : Prop

example : A → B :=
assume h : A,
show B, from ...
```

Once again, the ellipsis represents the argument for `B` using `A`. Notice that the hypothesis `A` is labeled with the letter `h`. We will see below how such labels can be used in a proof.

In a similar way, we teach students the standard patterns for dealing with conjunction, disjunction, negation, and universal and existential quantifiers. In each case, we use informal examples to motivate the corresponding rules in natural deduction, and later show students how to implement the patterns in Lean. The rule for eliminating an disjunction, corresponding to a proof by cases in ordinary mathematics, is a nice example. In natural deduction, the rule is represented as follows:

$$\dfrac{A \vee B \qquad \begin{array}{c}\overline{A}\\ \vdots \\ C\end{array} \qquad \begin{array}{c}\overline{B}\\ \vdots \\ C\end{array}}{C}$$

The rule is used to establish that a statement C follows from the assumption "A or B," denoted $A \vee B$ in symbolic logic. The rule has three premises: assuming, first, that we know that A or B holds, second, that we can prove that C follows from A, and, third, that we can prove that C follows from B, we can conclude that C holds outright. Here the lines over the local hypotheses A and B signify that these are only temporary assumptions, and that the resulting proof of C does not depend on them. In Lean, proof by cases is carried out as follows:

```
example : C :=
have h : A ∨ B, from ...,
or.elim h
  (assume h1 : A,
    show C, from ...)
  (assume h1 : B,
    show C, from ...)
```

This example is only schematic; in actual use, `A`, `B`, and `C` would likely be compound formulas, and ambient assumptions or previously established facts would allow us to fill in the ellipses with actual proofs. Notice that `h` labels the fact that `A ∨ B` is available in the current context, and `or.elim h` tells Lean what we wish to use the or-elimination rule described in symbolic terms above. The terminology matches the conventions used to name rules in natural deduction, solidifying the relationship in students' minds.

The following natural deduction proof establishes the implication $A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)$:

$$\dfrac{\dfrac{\dfrac{A \wedge (B \vee C)}{B \wedge C}\,{\scriptstyle 1} \qquad \dfrac{\dfrac{\dfrac{A \wedge (B \vee C)}{A}\,{\scriptstyle 1} \qquad \dfrac{}{B}\,{\scriptstyle 2}}{A \wedge B}}{(A \wedge B) \vee (A \wedge C)} \qquad \dfrac{\dfrac{\dfrac{A \wedge (B \vee C)}{A}\,{\scriptstyle 1} \qquad \dfrac{}{C}\,{\scriptstyle 2}}{A \wedge C}}{(A \wedge B) \vee (A \wedge C)}}{A \wedge B) \vee (A \wedge C)}\,{\scriptstyle 2}}{A \wedge (B \vee C) \rightarrow (A \wedge B) \vee (A \wedge C)}\,{\scriptstyle 1}$$

Students were expected to be able to carry out proofs like this for homework or on an exam. The labels on the hypotheses are used to tag the places in the proof where those hypotheses are canceled. Here is a corresponding proof in Lean:

```
variables A B C : Prop

example : A ∧ (B ∨ C) → (A ∧ B) ∨ (A ∧ C) :=
assume h : A ∧ (B ∨ C),
have h1 : A, from and.left h,
have h2 : B ∨ C, from and.right h,
or.elim h2
  (assume h3 : B,
    have h4 : A ∧ B, from and.intro h1 h3,
    show (A ∧ B) ∨ (A ∧ C), from or.inl h4)
  (assume h3 : C,
    have h4 : A ∧ C, from and.intro h1 h3,
    show (A ∧ B) ∨ (A ∧ C), from or.inr h4)
```

Students were also expected to be able to carry out proofs like this on their homework. (We gave ordinary pen-and-paper exams, so their ability to use Lean was only assessed on homework assignments.) The lines that begin with `have` support forward reasoning, establishing facts that can be used later on. The keyword `show` is usually unnecessary, but allows the user to display the conclusion of the proof that follows, making the proof more readable and robust.

As with assertions, the correspondence between natural deduction proofs and proofs in Lean is not exact, and we have to teach students how to translate their intuitions from one to the other. Thus, once again, the two representations provide alternative perspectives on informal mathematical proof and reinforce one another. The syntax is precise and unforgiving, and students often struggle with it. But most also come to enjoy it in the end. Using a proof assistant offers instant gratification: when a proof is correct, students know it right away, and feel good about it.

4 Formal perspectives on mathematics

It may not seem surprising that an interactive theorem prover can be useful when it comes to teaching the syntax and rules of logic. But after a few weeks in our course, logic is relegated to the background, and we turn to the fundamental building blocks of mathematics. We found that Lean continues to be a helpful tool in that respect as well.

We begin with a conventional treatment of sets, covering not just binary operations like union and intersection, but indexed unions and intersections as well. When we initially designed the course, an aspect of Lean's underling foundational framework gave us pause. In ordinary mathematical discourse, it is permissible to say "let A, B, and C be any three sets," without specifying what sort of elements they contain. In practice, mathematicians generally deal with sets of objects of some domain—for example, sets of numbers, sets of points, or sets of elements of some group. In Lean, every object is assumed to live in some such domain, its *type*, and it is most natural to introduce the concept of a set of elements of a particular type. In other words, in a Lean formulation, one can say "let A, B, and C be sets of natural numbers," or, more abstractly, "let U be some type, and let A, B, and C be sets of elements of U."

On the informal side, we were committed to following conventional mathematical terminology, making the treatment in our text seem as run-of-the-mill as possible. When we first designed the

course, we worried a good deal about how to smooth over the mismatches between our ordinary mathematical presentations and the formal representations in Lean. As it turns out, we needn't have been concerned. It is easy to explain to students that whereas the broadest conception of set allows mathematicians to consider the three-element set consisting of the number seven, the function which maps any real number to its square, and the president of France, the representation in Lean is more restrictive. Students know that informal language is not the same as a formal language, and can appreciate the design decisions that are made when deciding how to represent informal proof in formal terms.

An ordinary mathematics textbook might offer the following example of a proof of a set theoretic identity:

Theorem 4.1. *Let A, B, and C be any three sets. Then $(A \setminus B) \setminus C = A \setminus (B \cup C)$.*

Proof. Suppose x is any element of $(A \setminus B) \setminus C = A \setminus (B \cup C)$. Then x is in $A \setminus B$ but not C, and hence x is in A but not B. But this means that x is in A but not B or C, and hence not in $B \cup C$. So x is in $A \setminus (B \cup C)$, as required.

Conversely, suppose x is in $A \setminus (B \cup C)$, ... □

A corresponding proof in Lean is not as concise. Below we break out the implicit inference used above that if x is not an element either B or C then it is not an element of $B \cup C$, and make it an independent lemma. Modulo that lemma, the formal proof follows the structure of the informal one closely.

```
import data.set
open set

variables {U : Type}

lemma ex41a {A B : set U} {x : U} (h1 : x ∉ A) (h2 : x ∉ B) : x ∉ A ∪ B :=
assume h3 : x ∈ A ∪ B,
or.elim h3
  (assume h4 : x ∈ A,
    show false, from h1 h4)
  (assume h4 : x ∈ B,
    show false, from h2 h4)

theorem ex41 (A B C : set U) : (A \ B) \ C = A \ (B ∪ C) :=
eq_of_subset_of_subset
  (assume x : U,
    assume h1 : x ∈ (A \ B) \ C,
    have h2 : x ∈ A \ B, from h1.left,
    have h3 : x ∉ C, from h1.right,
    have h4 : x ∈ A, from h2.left,
    have h5 : x ∉ B, from h2.right,
    have h6 : x ∉ B ∪ C, from ex41a h5 h3,
    show x ∈ A \ (B ∪ C), from ⟨h4, h6⟩)
  ...
```

In the current implementation of the system, students are required to `import` the relevant theory and `open` the `set` namespace. The latter allows users to write `eq_of_subset_of_subset` rather than

`set.eq_of_subset_of_subset` to name the relevant lemma. We try to shield students from implementation details like these by providing them with templates on the homework assignments that provide such information. For example, we might carry out all the relevant imports, declare some variables, offers some examples of the background theorems we expect them to use, state a theorem, and then leave them the task of filling in the proof. This keeps the focus of the course on learning how to construct the proofs rather than learning how to use the theorem prover and its libraries.

Formal languages are equally useful in clarifying the logical structure of common properties of relations and functions. For example, after introducing the notions of injectivity, surjectivity, and bijectivity, we ask students how they can be represented in terms of symbolic logic. By this point, students are generally comfortable making the translation, and so the corresponding definitions in Lean come as no surprise:

```
variables {X Y Z : Type}

def injective (f : X → Y) : Prop := ∀ {x₁ x₂}, f x₁ = f x₂ → x₁ = x₂

def surjective (f : X → Y) : Prop := ∀ y, ∃ x, f x = y

def bijective (f : X → Y) := injective f ∧ surjective f
```

In Lean, curly braces around arguments like x_1 and x_2 indicate that they are to be left implicit when writing expressions, because they can typically be inferred from the context of the expression and other arguments. With these definitions, it is not hard to prove that the composition of two injective functions is injective, and similarly for surjective functions:

```
theorem injective_comp {g : Y → Z} {f : X → Y}
    (Hg : injective g) (Hf : injective f) :
  injective (g ∘ f) :=
assume x₁ x₂,
assume : (g ∘ f) x₁ = (g ∘ f) x₂,
have f x₁ = f x₂, from Hg this,
show x₁ = x₂, from Hf this

theorem surjective_comp {g : Y → Z} {f : X → Y}
    (hg : surjective g) (hf : surjective f) :
  surjective (g ∘ f) :=
assume z,
exists.elim (hg z) $
assume y (hy : g y = z),
exists.elim (hf y) $
assume x (hx : f x = y),
have g (f x) = z, from eq.subst (eq.symm hx) hy,
show ∃ x, g (f x) = z, from exists.intro x this
```

The use of the keyword `this` is another syntactic gadget in Lean: users can omit the label on a hypothesis, in which case `this` refers to the most recent hypothesis that has been left unlabeled. We introduce tricks like these casually, as side remarks. Some students enjoy using them, but they are by no means essential to carrying out the tasks that we assign for homework.

By this point, the gap between the kinds of facts we ask students to establish with pen-and-paper proofs and the kinds of facts we ask students to establish in Lean has grown. Full-blown interactive theorem proving can be a tedious and difficult affair, and it is a subject unto itself. In

our course we are committed instead to making sure that students are capable of writing the kinds of proofs that they would be expected to write in any course on the fundamentals of mathematical proof, and so we use only small examples in Lean that illuminate the main concepts and patterns of argument. For example, to illustrate proof by induction, we show students how to write a recursive foundational specification of addition on the natural numbers in terms of zero and the successor operation. We can then prove things like the commutativity of addition in Lean:

```
theorem add_comm (m n :  ℕ) : m + n = n + m :=
nat.rec_on n
  (show m + 0 = 0 + m, by rewrite [add_zero, zero_add])
  (assume n,
    assume ih : m + n = n + m,
    show m + succ n = succ n + m, from calc
      m + succ n = succ (m + n) : by rewrite add_succ
             ... = succ (n + m) : by rewrite ih
             ... = succ n + m   : by rewrite succ_add)
```

In contrast, when treating induction informally, we provide more interesting examples of combinatorial and number-theoretic identities, and discuss more general forms of induction. Ultimately, we would like to include some examples of these in Lean as well. But, at the moment, the material we have is more than sufficient to fill a one-semester course, and getting students used to the formal overhead may be too much of a distraction from our main goals.

5 Results

We have taught this course for three years now in the Dietrich College of Humanities and Social Sciences at Carnegie Mellon University, developing the materials along the way. The course is listed as a 200-level course, signaling that it is appropriate for first- and second-year undergraduate students but more advanced than courses at the 100 level. It fills a mathematical modeling breadth requirement for the college that can also be filled with the a calculus or statistics course. The course draws students from a wide range of backgrounds, from majors in mathematics, computer science, or physics to majors in philosophy or even business administration. Students at Carnegie Mellon are generally strong and not averse to mathematics and computer science, but the course does not presuppose any background beyond ordinary high-school mathematics.

Students rated the course highly in their evaluations. Each year we also solicited informal feedback both either verbally or with short questionnaires, and in the second year we enlisted the aid of Carnegie Mellon's Eberly Center for Teaching Excellence to carry out a more formal mid-semester evaluation. The classes were small and we have not accumulated precise data or developed quantitative measures of student improvement, so the data we report here is only anecdotal.

Students generally enjoyed the course, and some were quite enthusiastic. ("This was my favorite class to attend this semester—I always looked forward to it, and left happy.") On questionnaires we identified the various components of the course—symbolic logic, ordinary mathematics, and Lean—and asked students to compare them in terms of difficulty and interest. Students generally felt that the material was appropriately balanced. Some expressed a slight preference for one component over another, but among these students the specific preferences were fairly evenly distributed. Most importantly to us, students found the combination to be natural and helpful, and did not find it confusing when we asked them to switch between the different perspectives on proof.

Lean's syntax takes getting used to, and students were sometimes frustrated when the error messages were not sufficient for them to figure out what they were doing wrong. But all of them got the hang of it eventually, and many really enjoyed it. Using an interactive theorem prover offers instant gratification when a proof is accepted: Freek Wiedijk once described the feeling as being "like clearing a screen in a video game, but better." One student told us that she worked on unassigned problems in Lean in order to procrastinate writing English essays. Many students reported that, if anything, writing ordinary mathematical proofs was harder than writing formal proofs in Lean: at least with Lean they knew what the rules were, whereas they could not anticipate how their informal proofs would be received by person grading them.

The juxtaposition of formal and informal language led to interesting classroom discussion. Students were interested to discover that from a logical perspective the locution "A, but B" functions the same as "A and B" and the locution "A unless B" functions the same as "A or B." This led them to reflect as to what information is conveyed by the natural language variants. Some of our discussions focused on the ability of symbolic logic to clear up ambiguity, for example, by distinguishing between an inclusive and exclusive "or." We also discussed the fact that the phrase "everybody loves somebody" can be interpreted ambiguously as asserting that for every person there is another person they love, or that there is a single person that everybody loves. It is then interesting to see how the symbolic use of quantifiers resolves the ambiguity. Students got used to relativizing quantifiers, that is, translating phrases like "every car is red" and "some car is red" into the language of first-order logic.

Because we asked students to prove ordinary set-theoretic identities right after we completed the chapters on first-order logic, their natural tendency was to use symbols like quantifiers and arrows in their informal proofs. We explicitly told them not to do that, on the grounds that ordinary mathematical language favors using the words "every," "some," and "if... then" instead. Students were at first surprised and even skeptical of the claim that books in papers in ordinary mathematics, outside of formal logic, rarely ever use the logical symbols. After all, they are acutely aware that symbolic notation is absolutely central to mathematics. This led to interesting (and speculative) discussions as to why natural language is used for the logical connectives. Perhaps it can be attributed to the vagaries of linguistic convention, but perhaps it is because we find it easier to follow arguments verbally, as we recite them to ourselves. We also discussed the ways that the use of natural language can signal nuances, such as the use of the word "but" in a proof to signal either something unexpected, a contradiction, a fact that somehow complements a statement that has come right before, or the end of a proof.

The juxtaposition of formal proof and informal proof also facilitates another important discussion. In a formal proof it is clear that every step has to follow one of the given rules, but how much detail is needed in an informal proof? This highlights the fact that the purpose to writing an informal proof is not just to be correct, but to convince someone else of the validity of a result, and help them understand *why* it is true. Of course, these issues can also be discussed in an ordinary mathematics class, but the contrast with formal proof helps make the differences salient and meaningful.

We plan to continue developing the textbook, for example, with a chapter on algebraic structures and another on probability. More interestingly, as Lean evolves and gains new libraries and functionality, we plan to extend its use as well. We will use our pedagogical goals as a guideline: anything that provides useful insight into the nature mathematical proof is well worthwhile, whereas things that serve as a distraction from that goal are best avoided.

As we have taught it, the course covers a lot of ground. It requires commitment from the

students, a general facility with mathematics, and comfort with computer languages and their precise syntactic requirements. It could easily be expanded to a longer course, which would allow for a more leisurely pace, greater depth, and the inclusion of additional topics.

Our course aims primarily to prepare students for higher-level mathematics courses, introducing them to the definition-theorem-proof style of presentation, and making them consciously aware of mathematical norms and expectations of rigor. At the same time, the course prepares students for more advanced courses in logic: by the time they are done, they are comfortable speaking in the language of first-order logic, which is to say, reading, writing, and interpreting formal expressions. This puts them in a good position to appreciate metatheoretic results about syntax and semantics, including results regarding provability, definability, and completeness. Finally, the course offers good preparation for branches of computer science that invoke logical notions, including database theory, automated reasoning, and formal verification. In a number of branches of that discipline, facility with formal languages, formal rules, and semantic notions is essential.

We are by no means the only ones to use software to teach mathematical proof. For example, Daniel Velleman offers an online Java applet for writing proofs in elementary set theory to accompany his popular introductory text, *How to Prove It* [5]. The *AProS* project [4], also at Carnegie Mellon, uses an interactive proof tutor and allows students to write proofs in first-order logic and elementary set theory. Nathan C. Carter and Kenneth G. Monks are developing a mathematical word processor, *Lurch* [2], that can track the logical structure of a proof and verify inferences using back-end automation tools. They have used Lurch successfully in proof-based mathematics courses. In all the examples just cited, the approaches and technologies are slightly different from ours, but we expect that the pedagogical benefits are largely the same as the ones reported here. The goal of this article is just to share our own experience and observations, and to add our voices to the mounting chorus in support of using such tools to teach students how to read and write mathematical proofs.

References

[1] Jeremy Avigad, Robert Y. Lewis, and Floris van Doorn, *Logic and Proof*. Available online: http://leanprover.github.io/logic_and_proof/.

[2] Nathan C. Carter and Kenneth G. Monks, "From Formal to Expository: Using the Proof-Checking Word Processor emphLurch to Teach Proof Writing," in Schwell, Steurer, and Vasquez, "Beyond Lecture: Techniques to Improve Student Proof-Writing Across the Curriculum," MAA Press, 2016. Lurch project page: http://lurchmath.org/.

[3] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer, "The Lean Theorem Prover." In Amy P. Felty and Aart Middeldorp, eds., *Proceedings of the 25th International Conference on Automated Deduction (CADE-25)*, Springer, Berlin, pages 378–388, 2015. Project web page: https://leanprover.github.io/.

[4] Wilfried Sieg, "AProS Project: Strategic Thinking and Computational Logic." *Logic Journal of the IGPL* 15(4):359–368, 2007. Project web page: http://www.phil.cmu.edu/projects/apros.

[5] Daniel Velleman, *How to Prove It: A Structured Approach*, second edition. Cambridge University Press, Cambridge, 2006. Proof Designer: https://app.cs.amherst.edu/~djvelleman/pd/pd.html.