Teaching mathematics using Lean and controlled natural language

Patrick Massot
Université Paris-Saclay, France
Carnegie Mellon University, USA

Abstract

We present Verbose Lean, a library using the flexibility of the Lean programming language and proof assistant to teach undergrad mathematics students how to read and write traditional paper proofs. After explaining our main pedagogical goals, we explain how students use the library with varying levels of assistance to write proofs that are easy to transfer to paper because they look like natural language. We then describe how teachers can customize the student experience based on their specific pedagogical goals and constraints. Finally we describe some aspects of the implementation of the library, emphasizing how new aspects of the very recently released version 4 of Lean allow a lot of flexibility that could benefit many new creative uses of a proof assistant.

2012 ACM Subject Classification Human-centered computing → Natural language interfaces; Applied computing → Interactive learning environments; Theory of computation → Logic and verification

Keywords and phrases mathematics teaching, proof assistant, controlled natural language

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Funding *Patrick Massot*: Partially funded by the Hoskinson center for formalized mathematics

Acknowledgements The library described here is the result of work spread over five years and benefited from many interactions. Marie-Anne Poursat made it possible by trusting me to teach using a proof assistant. Frédéric Bourgeois and Christine Paulin-Mohring later joined me in teaching this course and discussing how to make it more useful. Our students also participated with their patience, questions and enthusiasm. Very recently, Julien Narboux, Pierre Boutry and Evmorfia-Iro Bartzia started using this library with students and made valuable comments. Simon Hudon started my Lean meta-programming education. It was then continued by Mario Carneiro and Kyle Miller who both also directly contributed crucial code that I would have been incapable of writing. Wojciech Nawrocki helped me a lot with widgets, adding several features to his ProofWidgets library for the needs of this project. More generarlly many people from the Mathlib community contributed indirectly by formalizing basic mathematics and creating tactics that we built on. I also benefited from conversations about teaching using a proof assistant, particularly with Heather Macbeth, Jim Portegies and Jelle Wemmenhove. Most of the work on the Lean 4 version was made possible by Jeremy Avigad's invitation to benefit from some of Charles Hosinkson's generous donation to CMU. And of course none of this would exist without the amazing work of Leonardo de Moura and Sebastian Ullrich on Lean 4.

1 Introduction

The transition from high-school mathematics to proof-based university mathematics is a well-known challenge for students. In the recent past, there have been several experiments using proof assistants to help students in this transition [2, 7, 12, 1, 17]. Here we really mean courses that consider the proof assistant only as an intermediate tool, not as a final goal. Note this tool is applicable to any kind of mathematics in principle but this account and the library it describes are biased towards elementary real analysis which is used in France as


© Patrick Massot;
licensed under Creative Commons License CC-BY 4.0

the main introduction to rigorous proofs.

Verbose Lean[1] is a teaching library and meta library for the Lean programming language and proof assistant. Lean is due mostly to Leonardo de Moura at Microsoft Research and then Amazon Web Service and the Lean FRO[2]. This library has three main layers. The bottom one is a set of tactics (i.e. proof producing programs) mimicking the granularity of proofs on paper. The middle one is a controlled natural language syntax whose goal is to ease the transition to paper proofs, at the cost of being slightly more difficult to write. The third layer is made of mechanisms that help students to write proofs by suggesting potential next steps. All three layers are heavily customizable, even without programming knowledge, and everything exists in French and English and is translatable to other languages.

This paper is intended for mathematics teachers and for people who want to see examples of using Lean's flexibility. It is organized roughly in order of decreasing importance. We first explain our pedagogical goals, then describe the student experience, then the teacher experience before concluding with some aspects of the implementation. That last section is more technical but could be useful to anyone interested in what can be done with Lean.

2 Main pedagogical goals

The first main goal is to train students to have a crystal clear view of the current state of the proof: what is currently being proven, what are the current assumptions, which mathematical objects are fixed. The next pedagogical goal is to train students to automatically perform proof steps depending of the syntactic structure of the goal. For instance a direct proof of a universally quantified statement starts with fixing an object of the relevant type.

A basic requirement here is to make sure that every statement has a clear status: is it something that is known to be true? Something that we assume? Something that will be proven soon? We claim that this goal is much easier to achieve if there is a really clear separation between stating, proving and using. For any given logical operator or quantifier, there are syntactic rules that explain how to form a mathematical statement it and some previously existing statements. Then there are rules that explains how to prove such a statement. Finally there are rules about how to use such a statement. This distinction seems obvious but in practice it is incredibly blurred, both by students and by professional mathematicians. Of course the difference is that mathematicians know what they are doing even when they are very sloppy about this distinction. One of the most frequent cases is failing to distinguish between stating the existence of a mathematical object satisfying some condition and fixing a witness. An example would be to say or write "since f is continuous and ε is positive, there exists δ such that…" and then refer to some δ on the following line. The other very common one is more tied to using symbols instead of text. It uses the implication symbol as an abbreviation of the word "hence" or "therefore", and more generally mixing using an implication and stating one. An example would be writing on a blackboard "f is polynomial $\implies$ f is continuous" instead of "f is polynomial hence continuous". This misuse of a symbol is extremely problematic since this alternative meaning of the symbol turns every legitimate use into nonsense. For instance reading the definition of convergence of a function f towards a number l at some point x_0 with this interpretation for the implication symbol gives "for every positive ε, there exists some positive δ such that for every x, $|x - x_0| < \delta$ hence $|f(x) - l| < \varepsilon$". A less ubiquitous but still common example is a

[1] https://github.com/PatrickMassot/verbose-lean4
[2] https://lean-fro.org/

confusion between stating a claim starting with a universal quantifier and beginning a proof of such a statement.

All those logical errors (or at least sloppy writing) also impact achieving a third pedagogical goal which is to train students to make a clear distinction between bound and free variables. This is of course very related to keeping track of what is fixed in the current state of the proof, but with the additional twist that some variable name can appear simultaneously as the name of a free variable and as a bound variable in some assumption or in the current target statement.

The next teaching goal that we want to achieve with this technology is to train students to classify proof steps into safe reversible steps that require no initiative and risky irreversible steps that require some creativity. The alternation between the routine safe steps and the risky creative steps is a crucial aspect of mathematical proof creation which is not necessarily emphasized when presenting a proof on a blackboard.

On top of that there is a final goal which is to learn how to choose indirect strategies instead of simply following the syntax of the current target. Those includes stating and proving an intermediate fact, using a lemma instead of reproving everything, and the use of the excluded middle axiom, e.g. through proofs by contradiction or contraposition.

The usual interfaces of proof assistants have many qualities that help achieving those goals. We will use the Lean terminology, but that part of the discussion applies equally well to Coq for instance. Near the beginning of proof of sequential continuity from continuity, the proof assistant can display something like:

```
f: ℝ → ℝ
u: ℕ → ℝ
x₀: ℝ
hu: u converges to x₀
hf: f is continuous at x₀
ε: ℝ
ε_pos: ε > 0
δ: ℝ
δ_pos: δ > 0
hδ: ∀ (x : ℝ), |x - x₀| ≤ δ ⇒ |f x - f x₀| ≤ ε
⊢ ∃ N, ∀ n ≥ N, |(f ∘ u) n - f x₀| ≤ ε
```

This display is called the *tactic state*. Most lines described mathematical objects that are fixed in the proof and assumption. The last line shows the current goal.

This is already tremendously useful to students, and completely impossible to emulate on a blackboard or in print. But there are also many challenges. The most obvious one is the need to learn the syntax of the software. In order to progress in proofs, one need to call *tactics* that are commands which usually do not look like mathematics but rather like programming. This is not a crucial problem, especially with students who also learn programming in other courses. But it does take time, so this issue prevents using a proof assistant on the side of a regular course with no dedicated time. A much more serious issue is that having a proof assistant syntax that is very different from paper proofs makes it a lot harder to transfer proving skills to paper. This is true in the direct case of transcribing a proof done on the computer but also in the longer run when students try to prove things on paper without writing a formal proof first.

A second challenge is to set up the right kind of automation. Traditional paper proofs are very far from mentioning every lemma that is used. There can be some value to write a couple of proofs mentioning every lemma. But doing this systematically is counter-productive with respect to the goals listed above. An extreme example would be lemmas asserting the

commutativity and associativity of addition and multiplication of numbers. More generally, things that are too obvious to be mentioned on paper should be done automatically by the proof assistant, whereas things that we want to see justified on paper should not. But this is an extremely vague and inconsistent specification. Powerful automation is also potentially problematic for students who have very slow compurters with little memory, and it can make error reporting more complicated. But it is crucial for the success of that kind of teaching use of proof assistants.

A last challenge that is less universal is that some countries do not use English as their main language. This is especially relevant for very young students. In France in particular, having anything written in English raises a huge barrier for undergraduate students.

3 Using the library as a student

3.1 Tactic language

In this section we will show what Verbose Lean looks like from the point of view of student users. We will show several possible variations but it is probably unwise to show all those variations to students. The following is a typical exercise solution.

```
Exercise "Continuity implies sequential continuity"
  Given: (f : ℝ → ℝ) (u : ℕ → ℝ) (x₀ : ℝ)
  Assume: (hu : u converges to x₀) (hf : f is continuous at x₀)
  Conclusion: (f ∘ u) converges to f x₀
Proof:
  Let's prove that ∀ ε > 0, ∃ N, ∀ n ≥ N, |f (u n) - f x₀| ≤ ε
  Fix ε > 0
  By hf applied to ε using that ε > 0 we get δ such that
    (δ_pos : δ > 0) and (Hf : ∀ x, |x - x₀| ≤ δ ⇒ |f x - f x₀| ≤ ε)
  By hu applied to δ using that δ > 0 we get N such that
    Hu : ∀ n ≥ N, |u n - x₀| ≤ δ
  Let's prove that N works : ∀ n ≥ N, |f (u n) - f x₀| ≤ ε
  Fix n ≥ N
  By Hf applied to u n it suffices to prove |u n - x₀| ≤ δ
  We conclude by Hu applied to n using that n ≥ N
QED
```

The first difference with the default syntax of Lean is that the statement clearly distinguishes the data, the assumption and the conclusion. Then each line of the proof sounds like natural language, but it is actually as rigid as any programming language. Remember the goal of this language is not to be easier to write, it is to be easier to transfer to paper.

The first line is completely optional, it only unfolds a definition. The second line shows how bounded quantifiers are handled by the library. The core logic of Lean does not involve those quantifiers. Given a predicate P, say on real numbers, the statement $\forall \varepsilon > 0, P(\varepsilon)$ is a notation for $\forall \varepsilon, \varepsilon > 0 \implies P(\varepsilon)$. In Verbose Lean, introducing a positive number can be done in one step (of course it can also be done in two steps). This generates a name `ε_pos` for the assumption that ε is positive. The next tactic, that spans the third and fourth lines above, uses that positivity but in a declarative way. However it does use the name `hf` that was assigned to the continuity assumption on f. A syntactic variant here would be to write

```
  Since f is continuous at x₀ and ε > 0 we get δ such that
    (δ_pos : δ > 0) and (Hf : ∀ x, |x - x₀| ≤ δ ⇒ |f x - f x₀| ≤ ε)
```

which only list claims and does not even explicitly mention ε before mentioning its positivity.

Another important style choice to make is the use of backward reasoning at the end, witnessed by the words "it suffices to prove". One could also replace the last two lines with

```
  Since ∀ n ≥ N, |u n - x₀| ≤ δ and n ≥ N we get h : |u n - x₀| ≤ δ
  Since ∀ x, |x - x₀| ≤ δ → |f x - f x₀| ≤ ε and |u n - x₀| ≤ δ we
    conclude that |f (u n) - f x₀| ≤ ε
```

which use both the nameless approach and forward reasoning only (those two aspects are independent, we gathered them only to prevent a combinatorial explosion of examples). The nameless approach is not purely stylistic, it also involves some implicit reasoning. For instance, with the same assumptions, we could write `since ε ≥ 0` and the library would silently derive this from $\varepsilon > 0$.

The example used so far only uses two definitions and the rules of logic. It features both using and proving statements formed using quantifiers and implication. Stating a universally quantified claim on line one and starting its proof on line 2 are very distinct operations. The first one involves the quantifier symbol while the second one involves the word "Fix".

The confusion between stating existence and extracting a witness is handled in a more subtle way. We could first state the existence using the symbol $\exists\,\delta$ and then have something like "Let us fix such a δ". Nothing prevents a teacher from implementing this syntax, but the trick of saying `we get δ such that` is a very nice compromise which distinguishes fixing a witness from merely stating existence and which stays very light to read.

The confusion between claiming an implication and using it is handled very simply. First the implication symbol is used only when stating. Then both backward and forward uses of implication mention both the implication and its premise. This applies both to the style referring to assumption names and to the nameless style.

Let us now consider another example: proving the squeeze theorem.

```
Example "The squeeze theorem."
  Given: (u v w : ℕ → ℝ) (l : ℝ)
  Assume: (hu : u converges to l) (hw : w converges to l)
          (h : ∀ n, u n ≤ v n) (h' : ∀ n, v n ≤ w n)
  Conclusion: v converges to l
Proof:
  Fix ε > 0
  Since u converges to l and ε > 0 we get N such that
    hN : ∀ n ≥ N, |u n - l| ≤ ε
  Since w converges to l and ε > 0 we get N' such that
    hN' : ∀ n ≥ N', |w n - l| ≤ ε
  Let's prove that max N N' works : ∀ n ≥ max N N', |v n - l| ≤ ε
  Fix n ≥ max N N'
  Since n ≥ max N N' we get (hn : n ≥ N) and (hn' : n ≥ N')
  Since ∀ n ≥ N, |u n - l| ≤ ε and n ≥ N we get
   (hNl : -ε ≤ u n - l) and (hNd : u n - l ≤ ε)
  Since ∀ n ≥ N', |w n - l| ≤ ε and n ≥ N' we get
    (hN'l : -ε ≤ w n - l) and (hN'd : w n - l ≤ ε)
  Let's prove that |v n - l| ≤ ε
  Let's first prove that -ε ≤ v n - l
  Calc -ε ≤ u n - l by assumption
       _  ≤ v n - l since u n ≤ v n
  Let's now prove that v n - l ≤ ε
  Calc v n - l ≤ w n - l  since v n ≤ w n
       _       ≤ ε        by assumption
QED
```

The beginning of the proof uses the same tactics as our first example. New things start with the line `Since n ≥ max N N′ we get (hn : n ≥ N) and (hn′ : n ≥ N′)`. Here we are using a lemma claiming that $n \geq \max(N, N')$ implies that $n \geq N$ and $n \geq N'$. But this lemma is not mentioned explicitly. The tactic saw that the claim $n \geq \max(N, N')$ is not a conjunction so it tried splitting it into the announced conclusions using so-called anonymous fact splitting lemmas. Here anonymous refers to the fact that their names do not appear in the proof script, but of course they actually do have names. The next two tactics (each spanning two lines) use the exact same mechanism using an anonymous lemma that split an inequality with shape $|x| \leq y$ into $-y \leq x$ and $x \leq y$.

The next tactic `Let's prove that |v n - l| ≤ ε` is completely optional, it recalls what is the current goal since it was never explicitly spelled out and we just went through three tactics that created new facts without changing current goal. This tactic could also have been used right after `Fix n ≥ max N N′`.

The next line is something new: `Let's first prove that -ε ≤ v n - l`. This tactic can be used to start a conjunction proof. But here the current goal is not a conjunction, it is turned into a conjunction by a so-called anonymous goal splitting lemma, which happens to be the converse the anonymous fact splitting lemmas used before (but those are completely separate lemmas from the framework point of view).

This tactic does a bit more than applying the lemma and splitting the resulting conjunction. Indeed we want to force students to announce the second part of the conjunction when the first one is proven. So instead of showing directly the second goal, the tactic state displays: `You need to announce: Let's now prove that v n - l ≤ ε` and refuses[3] any other tactic.

Returning to what happens during the proof of the first inequality, we see some computation introduced by the `Calc` word. This is based on the builtin Lean `calc` tactic, but the justifications are specific to our library. The first one in the example is `by assumption` which implicitly refer to the `hNl` assumption. A more explicit justification could be `from hNl`. What comes after the word `from` could also contain the words `applied to` and `using that` as in the third tactic of our first example. The next justification uses `since` which indicates a nameless approach: we claim that `u n ≤ v n` without explaining why; the tactic has to instantiate the assumption `h` to the free variable `n`. But there is more to it since this fact by itself is not sufficient to justify that calculation step. The tactic has to secretely invoke a lemma saying that $\forall x, y, z,\ x \leq y \implies x - z \leq y - z$. This is handled internally by calling the `gcongr` tactic of Carneiro and Macbeth.

Those two examples illustrate the main mechanisms that we use to get students to develop proof skills that are easier to transfer to paper than using the native Lean tactics. Of course they do not exhaust the list of tactics provided by our library. In particular there are tactics that allow to prove things by case disjunctions, using contraposition or proof by contradiction, or using the axiom of choice.

3.2 Assisted modes

The above examples can all be typed in the editor (typically VSCode to avoid teaching at the same time how to use Lean and a powerful editor such as vim or emacs). Lean then answers by updating the proof state and displaying error messages if needed. But mastering

[3] The courage to make this mandatory came from seeing it in the Coq waterproof project.

a lot of syntax is challenging, even if only one proof style is taught (for instance only the variants that do not use assumption names). So Verbose Lean offers two levels of assistance.

The fist level is the `help` tactic that can be used inside the proof. For instance, if the current target is `(f ∘ u) converges to f x₀` as at the beginning of our first example then the `help` tactics displays:

```
Help
· The goal starts with the application of a definition.
  One can make it explicit with:
  Let's prove that ∀ ε > 0, ∃ N, ∀ n ≥ N, |(f ∘ u) n - f x₀| ≤ ε
· The goal starts with "∀ ε > 0"
  Hence a direct proof starts with:
  Fix ε > 0
```

The above has two help messages and two suggestions. Clicking on a suggestion replaces the help tactic with the suggestion.

In the same example, there is a local assumption named `hu` saying that u converges to x_0. The answer to `help hu` is

```
Help
· This assumption starts with the application of a definition.
  One can make it explicit with:
  We reformulate hu as ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n - x₀| ≤ ε
· The assumption hu starts with "∀ ε > 0, ∃ N, ..."
  One can use it with:
  By hu applied to ε₀ using hε₀ we get N such that
    (hN : ∀ n ≥ N, |u n - x₀| ≤ ε₀)
  where ε₀ is a real number and hε₀ is a proof of the fact that ε₀ > 0
  The names N and hN can be chosen freely among available names.
```

Using this tactic with students suggests it already does a lot to tame the syntactic complexity of our controlled natural language. One could fear that students will constantly use it instead of analysing the goal or assumptions themselves, but this was not observed.

Especially in situations where there is not much time allocated to the use of a proof assistant, one can use a more assisted mode where proofs can be assembled at least partly through clicking. In this interaction mode, students click on expressions in the tactic state and get tactics suggestions in return. This subsumes the help tactic: when clicking on the full target or on a full assumption, the suggestions that given are the same that appear in the help command (assuming the default configuration is used). But one can also click on multiple assumptions, or on sub-expressions inside the target or inside an assumption.

For instance the example that proved sequential continuity from continuity can be done entirely by clicking. Clicking on the full goal suggests the first two lines of the proof. Then one needs to specialize the continuity assumption to the positive ε that was just introduced. This is done by clicking on the assumption and then clicking on ε. With this selection in the tactic state, one gets the following suggestions:

```
· By hf applied to ε using that ε > 0 we get δ such that (δ_pos : δ > 0)
    (hδ : ∀ (x : ℝ), |x - x₀| ≤ δ ⇒ |f x - f x₀| ≤ ε)
· By hf applied to ε / 2 using that ε / 2 > 0 we get δ such that (δ_pos : δ
    > 0) (hδ : ∀ (x : ℝ), |x - x₀| ≤ δ ⇒ |f x - f x₀| ≤ ε / 2)
```

Those two suggestions have the same shape but use either ε or $\varepsilon/2$ since specializing to half a given number is a very common move in elementary analysis and the default configuration is biased towards this kind of mathematics.

In the above example, the library does not check that it will be able to automatically prove the positivity side condition that appears after `using that`. This lets students judge the different suggestions.

4 Using the library as a teacher

4.1 Basic configuration

In this section we explain various mechanisms used for configuration which do not require programming expertise from teachers (of course a lot more is possible with programming). The goal is not to document every configuration possibility – since this is not a manual – but to show the configuration mechanisms that we use. We also show how this configuration depends on specific pedagogical goals, students expertise and time constraints.

The first decision to make is how much automation, if any, is desired when implicitly using lemmas. As explained in the previous section, there are two kinds of such lemmas, depending on whether they split a given fact or the current goal. For instance lemmas in the second category can be configured using `configureGoalSplitingLemmas Iff.into Subset.antisymm`. Listing a lot of lemmas in these commands could become very tedious. So we allow defining lemmas lists and referring to them in the configuration commands. We also pre-define some lists. Defining a list named `MyList` which contains the pre-defined list `LogicIntros` and the lemma proving set equality from double inclusion is done using

```
AnonymousGoalSplittingLemmasList MyList := LogicIntros Subset.antisymm
```

Those commands and all configuration commands are meant to be "hidden" in the teacher file. When changes are needed in the middle of an exercise file, one can use a macro. For instance `macro "switchConf" : command => `(configureGoalSplittingLemmas x)` would allow to simply to write `switchConf` between two exercises to avoid a long distracting line. Note that teachers will probably want to tell students about the list of anonymous lemmas, at least informally, to avoid creating confusion about what needs to be justified.

Tactics can also have configuration flags. For instance, say the goal is `¬ ∃ x:ℚ, x^2 = 2`. By default, starting the proof with `Assume for contradiction H : ∃ x : ℚ, x^2 = 2` will lead to the error message: "The goal is a negation, there is no point in proving it by contradiction. You can directly assume `∃ x : ℚ, x^2 = 2`". Teachers who fully embrace the confusion between a direct proof of a negation and a proof by contradiction, or simply need to focus on another battle, can use `allowProvingNegationsByContradiction`.

The next thing to configure is the assisted modes (help tactic and suggestion widget). First there are commands to disable those modes for teachers who want students to write everything by hand (in an exam setting, one can simply delete the relevant file for extra safety). Assuming they are enabled, many aspects are configurable. Each help message comes from a function and one can configure the available functions using the same kind of lists as with anonymous lemmas. For instance, in a basic course which progressively introduces different kinds of reasoning, one can disable messages suggesting a proof by contradiction in the beginning. One can also modify existing help functions with no programming knowledge by copy-pasting and editing only the text.

If the current lecture focusses on students knowing definitions then one can completely disable the help that unfolds definitions. One can also control in detail which definitions participate in unfolding suggestions. This has to be an opt-in mechanism to avoid having suggestions unfolding fundamental definitions such as the definition of real numbers or even the definition of addition of natural numbers. For instance the teacher file could contain:

`configureUnfoldableDefs continuous seq_limit` assuming the teacher library defined `continuous` and `seq_limit`.

The suggestion widget is also fully configurable. Really changing its behavior requires programming. But an easy tweak is to change how to produce data from the selection. We saw that selecting a real number ε and a universally quantified assumption h does not only propose to specialize h to ε but also to $\varepsilon/2$. The configuration for this can look like:

```
dataProvider mkSelf a := a
dataProvider mkHalf a := a/2
dataProvider mkMin a b := min a b
dataProvider mkMax a b := max a b
DataProviderList CommonProviders := mkSelf mkMin mkMax

configureDataProviders {
  ℝ : [CommmonProviders, mkHalf],
  ℕ : [CommmonProviders] }
```

In addition to a declaration list `CommonProviders` analogous to the one we use for anonymous lemmas, there are two new kinds of micro–DSLs (domain specific languages) here. First we define four "functions" with the `dataProvider` command which features no type information at all. Those are purely syntactic objects. They only participate in creating the widget suggestions on the syntactic level. Writing meaningless functions there would of course create trouble when accepting suggestions. Finally there is a JSON-like syntax in the `configureDataProviders` that registers data providers for different types. The goal of those DSL is to allow configuring this even for teachers who basically know nothing about Lean, maybe not even enough to write a function that can perform an algebraic operation either on natural numbers or on reals.

All the configuration options mentioned so far are specific to the Verbose library, but of course they come on top of the usual Lean configuration. A lot of the flexibility offered by Lean out of the box is very relevant to the kind of teaching targeted by our library.

This includes the whole parsing and elaboration pipelines. For instance Verbose Lean overwrites the notation for implication to use the double arrow symbol that is normally used in mathematics instead of a single arrow. The examples in this paper also use an infix notation for continuity and limits as in `u tendsto x`.

Overriding notation is not only about having a nice output. It also help mitigating unwanted side-effects of using type theory. For instance say we want to use the sequence of real numbers $n \mapsto 1/(n+1)$. Using its default configuration, Mathlib may need help to understand that $1/(n+1)$ is a real number and not a natural number. The correct interpretation can be enforced using a type ascription such as $1/(n+1:\mathbb{R})$. But this is distracting for students in the provided code, and failing to use such ascription in their own code can lead to inscrutable error messages. In this case it is much easier to override the meaning of the division symbol to always mean division or real numbers. One can also use a custom notation for function abstraction specialized to sequences of real numbers. For instance one can ensure `seq n ↦ ...` gets expanded to `fun n : ℕ ↦ (... : ℝ)`.

Note there is no way to completely avoid type ambiguities in the input without type ascriptions. We have seen students feeling the need to state as an intermediate fact something like `0 < 1`. Here there is no way Lean can guess whether this is meant as an inequality of real numbers or of natural numbers. And there is no way students can be aware of this issue without discussing the subtle status of the "inclusion" of $\mathbb{N}$ into $\mathbb{R}$. Lean will interpret the above statement as an inequality of natural numbers and our tactics will happily prove it.

But then using this intermediate fact will fail if the intended meaning was an inequality of real numbers. Note that Lean has an option, namely `pp.numeralTypes`, to always decorate literal numbers such as 0 or 1 when it displays them. This helps making to above problem easier to spot, but it does not fix the input issue and does not avoid discussing the subtlety.

4.2 Translating to a new language

The English language can be a huge barrier for undergraduate students. One can also imagine teachers who want to use a dialect of English. Verbose Lean comes with an English version and a French version. Adding a new language can be done by imitation without any knowledge of Lean programming. The process is to copy the English folder of the Verbose library and replace English words. In case of doubt, comparing the French and English versions can show the required modifications. For instance we see[4] in the English version:

```
declare_syntax_cat maybeApplied
syntax term : maybeApplied
syntax term "applied to " term : maybeApplied
syntax term "applied to " term " using " term : maybeApplied
syntax term "applied to " term " using that " term : maybeApplied

def maybeAppliedToTerm : TSyntax `maybeApplied → MetaM (TSyntax `term)
| `(maybeApplied| $e:term) => pure e
| `(maybeApplied| $e:term applied to $x:term) => `($e $x)
| `(maybeApplied| $e:term applied to $x:term using $y) => `($e $x $y)
| `(maybeApplied| $e:term applied to $x:term using that $y) =>
    `($e $x (strongAssumption% $y))
| _ => pure default

elab "We" " conclude by " e:maybeApplied : tactic => do
  concludeTac (← maybeAppliedToTerm e)
```

We will comment more on this code in the next section. Our point here is that we see a lot of mysterious things but understanding them is not required to write the French version:

```
declare_syntax_cat maybeAppliedFR
syntax term : maybeAppliedFR
syntax term "appliqué à " term : maybeAppliedFR
syntax term "appliqué à " term " en utilisant " term : maybeAppliedFR
syntax term "appliqué à " term " en utilisant que " term : maybeAppliedFR

def maybeAppliedFRToTerm : TSyntax `maybeAppliedFR → MetaM Term
| `(maybeAppliedFR| $e:term) => pure e
| `(maybeAppliedFR| $e:term appliqué à $x:term) => `($e $x)
| `(maybeAppliedFR| $e:term appliqué à $x:term en utilisant $y) => `($e $x $
    y)
| `(maybeAppliedFR| $e:term appliqué à $x:term en utilisant que $y) =>
    `($e $x (strongAssumption% $y))
| _ => pure default

elab "On" " conclut par " e:maybeAppliedFR : tactic => do
  concludeTac (← maybeAppliedFRToTerm e)
```

[4] The actual code has some more cases that were removed here for conciseness.

5 Some implementation mechanisms

The previous sections have been all about the pedagogical choices of the library, about how they can be tweaked by teachers and how students can use them. We now switch gears and turn to the question of the Lean mechanisms that allow all this. Many of those mechanisms are specific to Lean 4, the new family of versions of Lean that was officially released for the first time in September 2023 and puts flexibility of use in the center [5].

The flexibility of Lean as a proof assistant rests on two main pillars. The first one is that Lean is also a programming language and that almost all of Lean is implemented in Lean. This allows in particular to offer the possibility of overriding a lot of what Lean is doing, even pretty deep down, but we don't really use that directly. However we certainly use Lean as a programming language here, so we need a very quick introduction to that.

Lean is a pure functional programming language. So, fundamentally, it never does anything but defining and applying functions. However Lean makes extensive use of the monad pattern together with an extremely sophisticated notation system that can make it look a lot like imperative programming, but without the bad surprises [15]. An important inspiration is Haskell here. For our purposes, one can think of a monad `M` as a programming environment with a well specified state that can be read or written during program execution, a well specified way it can fail or not, and a well specified way of interacting or not with the outside world (such interactions could include reading files or printing things for instance). For any type α, the type `M` α is the type of programs that can do all this and return an element of type α (when they don't throw an exception if `M` includes the exception throwing capability). Running such a program requires passing in an initial state and actually also return the new state if the state includes writable parts.

An important example is the `CoreM` monad defined by Lean itself. It allows to describe and run programs that have read and write access to all definitions in scope, read only access to options, can fail by throwing certain kinds of exceptions, and can interact with the outside world (of course it has a more precise definition, we only give the flavor here).

On top of this `CoreM` monad sits the `MetaM` monad which mainly adds read and write access to the meta-variable context. A meta-variable is a place-holder that can be used in particular in a partially constructed proof. For instance at the very beginning of an interactive proof of a lemma, the full proof is a single meta-variable. By itself a meta-variable only stores a unique identifier. The meta-variable context is a data structure holding in particular for each meta-variable its expected type (that would be the conclusion of the lemma in our example) and its local context (that would be the assumptions of the lemma). On top of `MetaM` sits the `TacticM` monad which describes tactics, with additional access to all the relevant goals.

The second pillar of flexibility is the existence of typed concrete syntax objects as first class citizens [14, 13]. Here an important source of inspiration is the family of LISP languages, especially modern incarnations such as Racket. This is crucial for us. First it is crucial to our translation system that the syntax of tactics is clearly separated from their implementation. It also allows the assisted modes to provide suggestions that are guaranteed to be syntactically correct, because they produce syntax objects that are then printed as strings. This is seen in the snippets above that use syntax quotations such as `` `(maybeApplied| $e:term) ``. Syntax objects can also be turned into other syntax objects, either by macros such as the one showed in Section 4 or by functions in the library.

In order to explain this, we will follow part of the journey that allowed us to write in an earlier example: `By hu applied to δ using that δ > 0 we get N such that Hu :`

∀ n ≥ N, |u n - x₀| ≤ δ. The place where the corresponding syntax is hooked to the tactic implementation is

```
elab "By " e:maybeApplied " we get " news:newStuff : tactic =>
do obtainTac (← maybeAppliedToTerm e) (newStuffToArray news)
```

The elab command is a shortcut that allows to define a syntax in the first line and immediately assign it some meaning in the second line. On the first line we see two literal strings and two variables e and news. Those variables are also syntax objects with some syntax categories that are defined in Verbose Lean. They are based in the crucial syntax category term which is used for syntax representing Lean expressions. The (simplified) definition of maybeApplied representing functions that may be applied to arguments, and then a function turning such syntax objects into terms syntax objects have been seen in the first code snippet in Section 4.2. The first line of that snippet registers our syntax category and the next four lines describe four ways to build a syntax object in this syntax category. Those four ways are very simple and combine terms and literal words.

Now we can move to the second line of our elab command that calls the actual tactic. The do keyword starts a monadic program. Here it is a TacticM Unit program, i.e. a program in the TacticM monad that returns nothing – its only purpose is to manipulate the state of this monad. The type of the three involved functions are

```
obtainTac : TSyntax `term → Array MaybeTypedIdent → TacticM Unit
maybeAppliedToTerm : TSyntax `maybeApplied → MetaM (TSyntax `term)
newStuffToArray : TSyntax `newStuff → Array MaybeTypedIdent
```

We will ignore the third function. Note that the return type of the second one does not match the type of the first input to the first function. We need a term and not only a program computing a term in the MetaM monad. Fortunately, program in this monad can be used in the TacticM monad which extends it. And the arrow in (← maybeAppliedToTerm e) tells Lean to run the maybeAppliedToTerm e program and feed the result at this spot.

The definition of the maybeAppliedToTerm function was shown in Section 4.2. It is of course defined by pattern matching on the four possibilities to create a maybeApplied syntax object (plus a wild card possibility that is required because the syntax category could in principle be extended after the definition of this function). Note that the last interesting case also uses the strongAssumption% $y macro that expands to (by strongAssumption : $y) where strongAssumption is one of our tactics. Of course we could have used the expanded version here but the macro is used in several other places. Hence this example is a library function turning some syntax objects into other syntax objects using pattern matching and a macro that also does such a transformation.

The result type is not directly TSyntax `term but a program in the MetaM monad. This is because the quotation mechanism includes hygiene guarantees, a mechanism preventing accidental name capture and requiring some information from the surrounding context.

Note that one could inline this function into the obtainTac function. But this would make the latter into a language dependent function. As we saw in Section 4.2, Verbose Lean contains a French maybeAppliedFR syntax category and a function to turn syntax objects in this category into terms. It then uses the exact same obtainTac function from the language agnostic part of the library. Hopefully this already illustrate how we put to good use the monadic meta-programming framework of Lean and, crucially, its treatment of syntax objects. Syntax objects are also used to implement all the little DSLs we saw earlier.

We now want to discuss part of the implementation of obtainTac. Its first task is to turn the term it got as its first argument into a Lean expression, i.e. an abstract syntax

object whose type `Expr` is an inductive type whose main constructors correspond to the fundamental operations of lambda calculus: function application, function abstraction... This process is possible in the `TacticM` monad which has access to all definitions in scope as well as to the local context of the proof where this tactic is used.

Then `obtainTac` first tries to decompose the type of this expression. In our example this type is $\exists$ `N, ` $\forall$ `n` $\geq$ `N, |u n - x₀|` $\leq$ δ which can indeed be decomposed as a witness `N` and its property. This job of `obtainTac` is a trivial wrapper around a standard Lean tactic.

More interestingly, when such a decomposition does not make sense, the tactic will try to apply an anonymous splitting lemma. We saw already how to configure the lemmas that are tried. Now we want to discuss how this configuration is stored and updated. We saw that programs in the `CoreM` monad have access to existing declarations. More generally they have access to the so-called environment that stores declarations but also a lot more information. Lean allows to declare environment extensions storing user information for later use. In the case at hand, the stored information is simply a list of lists of declaration names. But our library also uses more interesting extensions for assisted modes.

The multilingual support for help and suggestions is based on a multilingual function dispatch framework by Mario Carneiro. Multilingual functions are first register it using the `register_endpoint` command. This gives a function that can immediately be used to define other functions. But running those functions requires implementing the endpoint in the current language, which is `en` by default but can be changed using `setLang`. For instance:

```
/-- Multilingual hello function. -/
register_endpoint hello : CoreM String

/-- Greeting function refering to our endpoint before any implementation
    is defined. -/
def greet (name : String) : CoreM String :=
  return (← hello) ++ " " ++ name

#eval greet "Patrick" -- throws error: no implementation of hello found
    for language en

implement_endpoint (lang := en) hello : CoreM String := return "Hello"

implement_endpoint (lang := fr) hello : CoreM String := return "Bonjour"

#eval greet "Patrick" -- returns "Hello Patrick"

setLang fr

#eval greet "Patrick" -- returns "Bonjour Patrick"
```

Note that above example creates three declarations: `hello`, `hello.en` and `hello.fr`, but only the first one is explicitly used. The implementations in this example are silly since they do no perform anything inside the `CoreM` monad, they simply return a value without reading or writing any `CoreM` state. But the `hello` function itself, which is created by the `register_endpoint` command, crucially uses `CoreM` to fetch the relevant information from the environment after reading the current language setting.

The way this is achieved illustrates an important point about Lean's flexibility. Lean as a proof assistant has very strong soundness guarantees, and the whole proof checking is handled by its type system. This translates to the default behavior of Lean as a programming language. We saw that being a pure functional programming language does not prevent

us from accessing state and having side-effects. One simply has to be honest about it by announcing in which monad we are working. Lean also allows to throw away type safety as long as we clearly announce it. And of course functions which do that cannot appear in proofs (they can participate in creating proofs, but can't appear in the end result).

A very simple version of the trick used in the multilingual framework is implemented in the example below. The `runFunctionOn` takes a string and a natural number, searches the environment for a declaration whose name is that string, then forcefully tells the Lean type system that this declaration is a function from natural numbers to natural numbers and applies it to the given number. The result is in `CoreM` $\mathbb{N}$ rather than $\mathbb{N}$ since it needs access to the environment to search for the relevant declaration and it could fail to find it so it needs to be able to throw errors – this is what happen in the second example below. But the new piece is the `unsafe` signpost in front of `def`. Indeed the declaration could be found but not with type $\mathbb{N} \to \mathbb{N}$, bringing us into undefined behavior territory. This is what happen in the third example below where a function concatenating strings is found.

```
unsafe def runFunctionOn (function : String) (a : ℕ) : CoreM ℕ := do
  let myFun ← evalConst (ℕ → ℕ) (Name.mkSimple function)
  return myFun a

def foo (a : ℕ) := 2*a + 1

#eval runFunctionOn "foo" 1 -- returns 3

#eval runFunctionOn "baz" 1 -- fails with error message: unknown
    declaration baz

def bar (a : String) := a ++ a

#eval runFunctionOn "bar" 1 -- crashes Lean
```

The moral is that programming in Lean allows to do very unsafe things that completely bypass the guarantees offered by the type system, but this must be clearly flagged and cannot be used as a proof (explaining how soundness is protected is beyond the scope of this discussion). Note that our actual multilingual dispatch is much more careful and checks that types match before calling functions so that teachers don't crash Lean when they make a type mistake in the implementation of a new help message. For performance reasons we don't want to perform this check at every function call, so we also use an extension that keeps track of a list of endpoints and type-checked implementations.

Although we insisted on our use of concrete syntax objects, we also use abstract ones, with type `Expr`, in the tactic backends. We even have a custom version `VExpr` with many more constructors that are useful when analysing goals and assumptions in assisted modes. For instance bounded quantifiers have dedicated constructors. We have a function parsing an `Expr` into a `VExpr`, hence factoring out work that many help functions would need to do. The type of help functions that analyse the goal is `MVarId` → `VExpr` → `SuggestionM Unit` where `MVarId` is used to indicate the relevant goal and the `SuggestionM` monad accumulates suggestions while providing access to the `MetaM` monad. Such functions are registered as part of the configuration by teachers, together with a pattern indicating (coarsely) which kind of goal they comment on. Calling the help tactic uses a discrimination tree to quickly locate functions with the relevant pattern and then check whether they are active in the configuration. This use of discrimination trees is not necessary until someone implements thousands of help functions, but the infrastructure is provided by Lean so it is free.

The last piece of Lean infrastructure that we want to comment on is the framework that allows us to build the suggestion widget. There are quite a few layers here. Lean implements the Language Server Protocol (LSP) with many extensions related to the so called info view which gather the tactic state display, various messages and user-defined widgets. Deep down, widgets are Javascript modules that export a React component that is displayed by the VSCode extension, can access information from Lean and modify the current document. However the ProofWidgets library [9] offers a powerful abstraction that allows us to ignore Javascript. In particular it features a JSX-like DSL as well as React components written in Lean and having a Lean interface. As a result, Verbose Lean does not contain a single line of actual Javascript or HTML. For instance the loop printing the suggestions is:

```
for ⟨linkText, newCode, range?⟩ in suggs do
  let p : MakeEditLinkProps := .ofReplaceRange doc.meta
    ⟨params.pos, params.pos⟩ (ppAndIndentNewLine curIndent newCode) range?
  children := children.push
    <li style={json% {"margin-bottom": "1rem"}}>
      <MakeEditLink
          edit={p.edit} newSelection?={p.newSelection?} title?={p.title?}>
        {.text linkText}
      </MakeEditLink>
    </li>
```

The `li` tag is directly turned into a HTML list item, whereas `MakeEditLink` refers to a ProofWidgets component. This component is rendered as a HTML link which, upon getting clicked, edits the proof script. All this is fully type-correct Lean code, with real-time typechecking and the expected editor support (for instance ctrl-clicking on MakeEditLink jumps to the relevant declaration).

The tactic state natively allows to select names or sub-expressions in the local context or the goal. It records this information as an array where each element inhabits an inductive type `Lean.SubExpr.GoalLocation` having a constructor for each kind of selection, for instance a constructor for en element of the local context, one for a sub-expression in the type of such an element, etc. This layout is not convenient for our purposes so we introduce another datatype `SelectionInfo` which gather the same information by type of selections. We also have many functions querying this information. Each suggestion provider has type `SelectionInfo → MVarId → WidgetM Unit` analogous to the help function type.

6 Related work

Both the dream of using proof assistants for teaching and the work on alternative interfaces are very common. However most teaching uses focus on computer science, logic or discrete mathematics, or even on proof assistants for themselves.

One notable exception is the work of Heather Macbeth at Fordham university [8]. However her course is more focused on computations and less on reasoning, so the need for a controlled natural language is less pressing. What is common to both contexts is the need for automation that is adapted to the level of details expected from students. And indeed some of our tactics rely on tactics develop for Macbeth's course.

Even more relevant is the comparison with the Coq-Waterproof project [16] that was developed independently and shares many goals with Verbose Lean. Discussing with its authors led to several improvements in our work. One thing that we still do not have is a nice custom text editor to mix rendered comments and interactive exercises. On the other hand, we do benefit from using a very flexible proof assistant that easily allows syntactic

freedom and interactive interfaces, as explained in this paper. The resulting proof scripts are closer to paper proofs and the user interaction model is richer.

Also very relevant and interesting is the Diproche system [3, 4]. Its focus on controlled natural language is even greater than in Verbose Lean. Proofs are sequences of assertions in a more flexible language. Those assertions are sent to an automated prover that complains if it cannot justify a step. The main downside is that the proof structure is much less clear.

On the topic of alternative interfaces, there are again many attempts that seem practicable only for pure logic. For instance this is the case of the Actema project [6] which proposes a drag and drop interface that is partly a more graphical version of our suggestion widget and can interface with Coq. However it seems difficult to integrate with computations as in our squeeze theorem example (which was chosen as the simplest example involving computations). And it very explicitly targets leaving no written trace at all, hence has a very different goal.

Also in the same category but explicitly targeting teaching very young university students is the d∀∃duction project [11]. It features a graphical interface based on selecting sub-expressions and clicking buttons, with a completely invisible Lean backend. Again there is no written trace and it seems difficult for students to transfer their new skills to paper.

Edukera [10] is another point and click interface that produces a written text. It is a web interface based on Coq. Its first main drawback is that teachers cannot write exercises or configure anything, they simply have access to a fixed set of exercises. In addition, there is no possibility to directly input text. The interface is only based on clicks and the text is purely on the output side. Also the development of Edukera stopped in 2018.

As far as we know, none of those very nice projects have multilingual support except for Edukera. Most of them are only in English, Diproche is only in German and d∀∃duction is only in French.

7 Future work

Although some version of this library has been used for five years in University Paris-Saclay at Orsay, it is very much a work in progress, especially since the switch to Lean 4 made it a lot more flexible. The suggestion widget in particular has not been used with students yet, and is bound to need refinements and extensions. Error reporting is also a never-ending work in progress. Each new interface or piece of automation requires more care in case of incorrect input, and students always find new ways to trigger unexpected error messages. On the multi-lingual side, one short-term goal is to make it easier to create variants of an existing language. Another project is to offer more exercises that are ready to use or modify for teachers. Existing exercises are not yet all ported to Lean 4, and new ones should be created in different fields of elementary mathematics.

On the pedagogical side, there are plan to rigorously assess the benefits of using this library with the APPAM[5] team which includes specialists in education sciences.

References

1 Jeremy Avigad. *Learning Logic and Proof with an Interactive Theorem Prover*, pages 277–290. Springer International Publishing, Cham, 2019. doi:10.1007/978-3-030-28483-1_13.

2 Evmorfia-Iro Bartzia, Emmanuel Beffara, Antoine Meyer, and Julien Narboux. Underlying theories of proof assistants and potential impact on the teaching and learning of proof.

[5] https://appam.icube.unistra.fr/

In *12th International Workshop on Theorem proving components for Educational software*, Rome, Italy, July 2023. Julien Narboux and Walther Neuper and Pedro Quaresma. URL: https://hal.science/hal-04227823.

3 Merlin Carl. Number theory and axiomatic geometry in the diproche system. In Pedro Quaresma, Walther Neuper, and João Marcos, editors, *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France, 29th June 2020*, volume 328 of *EPTCS*, pages 56–78, 2020. doi:10.4204/EPTCS.328.4.

4 Merlin Carl, Hinrich Lorenzen, and Michael Schmitz. Natural language proof checking in introduction to proof classes - first experiences with diproche. In João Marcos, Walther Neuper, and Pedro Quaresma, editors, *Proceedings 10th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2021, (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021*, volume 354 of *EPTCS*, pages 59–70, 2021. doi:10.4204/EPTCS.354.5.

5 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

6 Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. A drag-and-drop proof tactic. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 197–209. ACM, 2022. doi:10.1145/3497775.3503692.

7 Marie Kerjean, Frédéric Le Roux, Patrick Massot, Micaela Mayero, Zoé Mesnil, Simon Modeste, Julien Narboux, and Pierre Rousselin. Utilisation des assistants de preuves pour l'enseignement en L1. In *Gazette de la SMF*, volume 174, Octobre 2022.

8 Heather Macbeth. The mechanics of proof. https://hrmacbeth.github.io/math2001/.

9 Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An extensible user interface for Lean 4. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 24:1–24:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ITP.2023.24, doi:10.4230/LIPICS.ITP.2023.24.

10 Benoît Rognier. Edukera. https://edukera.com/.

11 Frédéric Le Roux. D∀∃duction. https://perso.imj-prg.fr/frederic-leroux/d∀∃duction/.

12 Athina Thoma and Paola Iannone. Learning about proof with the theorem prover Lean: the abundant numbers task. *International Journal of Research in Undergraduate Mathematics Education*, 8(1):64–93, Apr 2022. doi:10.1007/s40753-021-00140-1.

13 Sebastian Ullrich. *An Extensible Theorem Proving Frontend*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2023. URL: https://nbn-resolving.org/urn:nbn:de:101:1-20230802045824809330 72.

14 Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. *Log. Methods Comput. Sci.*, 18(2), 2022. URL: https://doi.org/10.46298/lmcs-18(2:1)2022, doi:10.46298/LMCS-18(2:1)2022.

15 Sebastian Ullrich and Leonardo de Moura. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proc. ACM Program. Lang.*, 6(ICFP):512–539, 2022. doi:10.1145/3547640.

16 Jelle Wemmenhove, Thijs Beurskens, Sean McCarren, Jan Moraal, David Tuin, and Jim Portegies. Waterproof: educational software for learning how to write mathematical proofs, 2022. arXiv:arXiv:2211.13513.

17 Xiaoheng Yan and Gila Hanna. Using the Lean interactive theorem prover in undergraduate mathematics. *International Journal of Mathematical Education in Science and Technology*, 0(0):1–15, 2023. arXiv:https://doi.org/10.1080/0020739X.2023.2227191, doi:10.1080/0020739X.2023.2227191.