

École polytechnique de Louvain

Proof of concept of an interactive theorem prover system using natural language input

Author: **Andres ZARZA DAVILA**

Supervisor: **François GLINEUR**

Readers: **François GLINEUR, Sébastien MATTENET, Peter VAN ROY**

Academic year 2020–2021

Master [120] in Data Sciences Engineering

Acknowledgements

I would first like to thank my supervisor Professor François Glineur for his guidance and knowledge throughout this project. I would also acknowledge assistant Sébastien Mattenet for his supervision and contributions. Both of your insights and ideas, greatly helped to shape the direction and form of this project.

I would also like to thank some of my colleagues in particular Nizar Bousselmi for your reviews and remarks.

In addition, I would like to thank my friends and family for their support as well as providing an empathetic ear. Guillaume Dieleman's remarks and advice were particularly useful for the concrete implementation.

Abstract

This work presents a proof of concept for a mathematical proof system that would allow users to write in a close to natural language while generating proofs that are machine verifiable. The main objective is to assess the feasibility and features of such a project and explore promising leads to this end. The current implementation is a web application written in Python/Javascript with backend in Django and frontend in React. It constructs and verifies proofs with Lean, an interactive theorem prover. Natural language processing is made by simple pattern matching. The current implementation is still somehow limited but allows to envision a more advanced system. The last chapter develops possible ideas to further improve and extend the current proof of concept.

Contents

1	Introduction	8
1.1	Context	9
2	Review of other systems	10
2.1	System for common mathematical knowledge	10
2.2	Proof assistants	11
2.3	Desired features for an ideal system	12
2.4	Verification delegation to Lean	16
2.4.1	Dependant type theory	17
2.4.2	Reasons for choosing Lean	19
2.5	Other related projects	19
2.5.1	Formal Abstracts	19
2.5.2	HolPy	20
2.5.3	Lean formatter	20
3	Implementation	21
3.1	Overview	22
3.1.1	Login/Create account	23
3.1.2	Write a proof	24
3.1.3	Write a statement and send it	30
3.1.4	Write a proof for a sent statement	34
3.2	System structure	36
3.2.1	Database	37
3.3	Translation	38
3.3.1	Context	39
3.3.2	Parsing approach	40
3.3.3	List of current possible translations	42
3.3.4	Method for increasing translation ability	44
3.4	Sandwich theorem example	45

4	Future steps and conclusion	50
4.1	Translation	50
4.1.1	Machine Learning	50
4.1.2	Community system	51
4.1.3	Lean insights	51
4.2	Mathlib integration	52
4.3	Structured proofs	52
4.4	Community	53
4.5	Conclusion	55
A	Screens from the implementation	56
A.1	Limit of a sum example	58

List of Figures

2.1	Lean example	11
2.2	Adapted proof of Spivak corollary of Mean Value Theorem from [5]	14
2.3	Adapted proof of Spivak corollary of Mean Value Theorem from [5] with less detail	15
2.4	Schematic interaction with external system	16
2.5	Type of a theorem indicating that the sum of two series converges to the sum of the respective limits of each series	17
2.6	sum_limit theorem	18
3.1	Login view	23
3.2	Create an account	24
3.3	Menu logged in	24
3.4	Interface to write a complete proof	25
3.5	Create a MathQuill input	26
3.6	MathQuill changing $\backslash\epsilon$ to its symbol	26
3.7	Hypotheses manipulation	27
3.8	Ask State at initial goal	28
3.9	Unrecognized hypothesis error - Failed to translate to Lean	28
3.10	Lean feedback indicates an error - r has not been declared	28
3.11	Proof line	29
3.12	Feedback indicating success of proof	30
3.13	List of owned theorem statements	30
3.14	Edition of a statement	31
3.15	Send statement to users 2 and 3	32
3.16	Remove some users from the list that received the statement	33
3.17	List of statements sent to a user	34
3.18	Prove a theorem with fixed statement	35
3.19	Translation structure	36
3.20	Database diagram	37
3.21	Proof lines stored in database	38
3.22	Language interface	39
3.23	By inequality properties - example	42

3.24	Sandwich Theorem - 1	45
3.25	Sandwich Theorem - 2	46
3.26	Sandwich Theorem - 3	47
3.27	Sandwich Theorem - 4	48
3.28	Sandwich Theorem - 5	49
4.1	Example of visual for structured proofs	53
A.1	Menu when not logged as a user	56
A.2	Confirmation of deletion of user from list of people having a statement	56
A.3	Confirmation of deletion of statement	57
A.4	Animation while awaiting server response	57
A.5	Ask to reset password	57
A.6	Limit of a sum - 1	58
A.7	Limit of a sum - 2	59
A.8	Limit of a sum - 3	60
A.9	Limit of a sum - 3	61

Chapter 1

Introduction

The 20th century has seen an increase of mathematical normalization and formalization as shown for example by the notorious Principia Mathematica[12]. The advancements in computer science and other fields have laid path for several proof assistants such as Coq, Isabel, Lean helping researchers in the difficult tasks of writing formal proofs. This desire for combining mathematical formalization and computer systems was highlighted by several researchers such as shown in the QED manifesto [2, 11]. In this early 21st century, digitization of the world and creation of smarter programs is still rapidly increasing. One can only hope that further progress in the fields of computer science, artificial intelligence, natural language processing will result in better software that will empower mathematicians to write proofs with friendly interfaces, smart suggestions, and leaving the tedious task of verifying proofs to automated systems. In this line of thinking, this thesis presents a proof of concept for a system that would allow to write proofs in natural language while making them machine verifiable.

The resulting implementation is available at two repositories :

- <https://github.com/azarzadavila/mapros>
- <https://github.com/azarzadavila/mapros-frontend>

The name of the repositories, *MaProS*, stands for Mathematical Proof System. This current system is a website application and uses the theorem prover Lean[3].

This work starts by presenting what would be the desired features of an ideal system. Then comes a chapter reviewing some existing systems and discussing reasons to include Lean. After that, the current implementation is discussed. Finally future possible steps are considered to improve and continue the system.

1.1 Context

Mathematical formalization has been a concern for many mathematicians in particular in the 20th century. It is guided by a desire of certainty and robustness of proofs. From the lens of formalism, mathematical proofs are generated by syntactic transformation of proven or admitted sentences to new sentences. In that sense as explained in [10], *The guiding idea behind formalism is that mathematics is not a body of propositions representing an abstract sector of reality but is much more akin to a game, bringing with it no more commitment to an ontology of objects or properties than ludo or chess.* The set of allowed transformation rules, called rules of inference is limited and clearly defined. By using these rules and stating set of accepted axioms, a formal proof is a pillar of truth and confidence in a mathematician's spirit. However, writing a formal proof does come with a price. It is a long tedious process that requires the strictest adhesion to a set of predefined rules. It usually involves much more writing than what is done in a research paper.

Today, mathematicians have proof assistants to help them in the quest of formalism. A proof assistant is a software which purpose is to generate formal proofs. The user writes the proof with a dedicated language of the proof assistant. A proof written in that way is machine verifiable. It means that the proof assistant can read the proof and detail it internally to verify if it does indeed respect all the rules of the chosen formalism. Proof assistants, also called interactive theorem prover, can by that mean help to verify a proof. They usually also include ways to not enter all details of a proof but for example to tell the machine to consider all cases by applying specific procedures. This eases the burden of writing a proof with a long sequence of rather easy steps. The first notorious theorem that was proved with a proof assistant is the four color theorem[1].

By their digital nature, proof assistants offer also new community advantages. With a well designed system, communities can collaborate to build growing sets of mathematical knowledge available online. Proof assistants are yet to include all mathematics in a formalized set of knowledge. Freek Wiedijk maintains a list of 100 theorems specifying in which proof assistant it has been proved or not yet. This list is available at the following link :

<https://www.cs.ru.nl/~freek/100/>

This thesis aims at exploring the possibility of closing the gap between writing a formal proof and a proof of a research paper style. It does this by introducing an additional translation step between the user and an interactive theorem prover.

Chapter 2

Review of other systems

In this chapter, some systems about the manipulation of mathematical proofs are discussed as well as how one of these systems, Lean, is included in the implementation presented. The considered aspects about working with proofs include

- writing
- reading
- verifying
- sharing
- reusing
- storing

Many systems exist for treating these different aspects. Some of those systems excel at some features but none do well in all at the same time.

2.1 System for common mathematical knowledge

In this part, let us consider the case of Wikipedia. I will suppose the reader has read many proofs on Wikipedia. Nonetheless, let us highlight some of the interesting features.

Wikipedia has a great input system. It allows writing in Latex which is great for mathematical purposes. With the used markup language one can write and structure a proof as it would be done in a research paper for example. It is possible to add links to other pages, for example to point another theorem.

Wikipedia also has the great advantage of being easily available to anyone from the web. Read access is free and knowledge is centralized. Discussions about topics are encouraged and anyone can suggest editions.

The main downside of Wikipedia is it does not have any kind of automatic proof verification. It can also allow for some inconsistencies between pages. For example, pages might differ by the axioms they consider.

2.2 Proof assistants

A proof assistant usually has two goals : help a user to construct a proof and verify it in a rigorous manner. The way these goals are achieved can vary a lot from one system to the other. Later on we will discuss in detail about Lean one such proof assistant that uses type theory[3]. Once something is proved using Lean it is close to absolute certainty that the proof is correct. Let us see what Lean looks like. Figure 2.1 shows an example of a proof in progress.

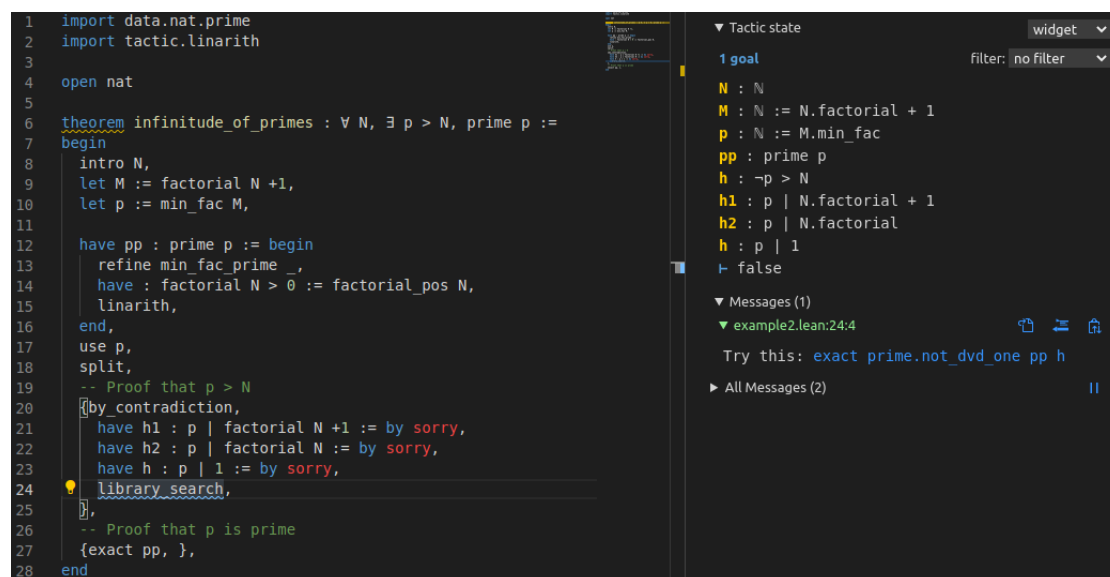


Figure 2.1: Lean example

The editor is Visual Studio Code that integrates a Lean plugin. This proof comes from the video available at

https://www.youtube.com/watch?v=b59fpAJ8Mfs&ab_channel=leanprovercommunity

It is an example of constructing a proof that there is an infinitude of prime numbers. The main idea of this proof is that if we take any natural number, N , we can build

a prime number greater than this natural. To do so, one can build M by taking the factorial of N and adding it one. The smallest prime factor of M is a prime number greater than N .

The first interesting feature is that the proof can be structured. This is visible by the presence of multiple begin-end as well as brackets. The proof is in construction as any part where *sorry* is written tells the compiler that this part is not yet proven but should be ignored for now.

The second feature to notice is the interactivity of the program. The position of the cursor is highly important as it indicates the current state of the proof at the specific location. A proof in Lean is usually made by a sequence of *tactics*.

Tactics

A tactic is a procedure that either tries to solve the goal or that somehow modifies the current state, for example by generating a new variable. The documentation for Lean tactics is available at :

<https://leanprover.github.io/reference/tactics.html>.

It can be viewed as a more advanced version of inference rules. Although this term comes from Lean, it will be used in the rest of this work on other contexts.

Some high level tactics ask for help for example *library_search* indicates to look in the library if there is a tactic that might help solve the goal. Here Lean tells us to try *exact prime.not_dvd_one pp h*. The interface is rather aesthetic in part by the use of colors and Unicode for mathematical concepts used regularly : $\forall, \exists, \dots$. Through this small example one can image the power of such software but also see that it does require some learning. The proof is not as readable as in a research paper and writing a proof in Lean requires an important prior knowledge.

2.3 Desired features for an ideal system

As discussed existing systems already possess many benefits. However both systems like Wikipedia and proof assistants also have some downsides. Wikipedia excels at readability and at collaborative work both in writing and access. However, it may allow for some inconsistencies between one page to another and does not have any kind of automatic proof verification. In contrast proof assistants such as the one mentioned are excellent for both verification and construction of the proof. However, such systems usually involved the use of a programming language. By using this unambiguous structured language, users communicate exactly what they want. There are several downsides involved with the lack of use of natural language.

The proof is usually much less readable than it would be for example in a research paper.

Now that we have seen disadvantages of existing systems let us explore the features that would represent an ideal system combining discussed benefits.

Target The ideal system would allow people in possession of mathematical knowledge to write a proof with a very quick learning curve. If experts in a given field want to write a proof they already know, it should take a time similar to writing it on a Latex editor. The proofs should be readable for anyone with enough mathematical understanding. More precisely, it should at least be easy for an undergraduate student to search for a theorem proof in the system and be able to read the proof as the student would do in a textbook. The system should also be accessible to anyone on the internet and allow for collaborative work.

Structure Proofs should be stored in the system and easy to lookup. A theorem statement should be separated from its proofs. This would allow to write different proofs for a given theorem. A given theorem should be of course reusable in order to build a growing set of knowledge. Ideally, one would be able to see on what other theorems a given theorem is built upon. This kind of feature would be great to understand how a given set of axioms defines a mathematical theory. In order to allow for great readability for both experts and more novice users proofs should be written in a hierarchical way. This is an idea explored in [5]. Figure 2.2 shows an example of such a structured proof.

If $f'(x) > 0$ for all x in an interval I , then f is increasing on I .

1. It suffices to assume

1. a and b are points in I
2. $a < b$

PROOF: By definition of an increasing function

2. There is some x in (a, b) with $f'(x) = \frac{f(b)-f(a)}{b-a}$.

2.1. f is differentiable on $[a, b]$.

PROOF: By 1.1, since f is differentiable on I by hypothesis.

2.2. f is continuous on $[a, b]$.

PROOF: By 2.1 and Theorem 1.

2.3. Q.E.D.

PROOF: By 2.1, 2.2, and the Mean Value Theorem.

3. $f'(x) > 0$ for all x in (a, b) .

PROOF: By 2 and 3.

4. Q.E.D.

PROOF: Assumption 1.2 implies $b - a > 0$, so 4 implies $f(b) - f(a) > 0$, which implies $f(b) > f(a)$. By 1, this proves the corollary.

Figure 2.2: Adapted proof of Spivak corollary of Mean Value Theorem from [5]

This kind of structure would allow a user to see or hide more detail. For example the step number 2 could be hidden to see less detail. The example with less detail is shown at Figure 2.3.

If $f'(x) > 0$ for all x in an interval I , then f is increasing on I .

1. It suffices to assume

1. a and b are points in I
2. $a < b$

PROOF: By definition of an increasing function

2. There is some x in (a, b) with $f'(x) = \frac{f(b)-f(a)}{b-a}$.

PROOF: By 1.1, 1.2, differentiability of f and the Mean Value Theorem.

3. $f'(x) > 0$ for all x in (a, b) .

PROOF: By 2 and 3.

4. Q.E.D.

PROOF: Assumption 1.2 implies $b - a > 0$, so 4 implies $f(b) - f(a) > 0$, which implies $f(b) > f(a)$. By 1, this proves the corollary.

Figure 2.3: Adapted proof of Spivak corollary of Mean Value Theorem from [5] with less detail

Input/Output To ease the learning curve and be more readable, most of interactions with the system should happen in natural language. That is, when writing a line of the proof, users should issue what they mean at the given line in natural language with the help of Latex/Unicode symbols and commands. The output of the system should also be as much as possible in natural language with mathematical symbols. The system should ideally be interactive. The construction of the proof may require some outputs from the system. A great feature would be that the display be configurable. For example, some people might want to see a matrix multiplication with a dot $\cdot$ while others might prefer a cross $\times$.

Verification The system should verify the proof and specify if it is correct. The verification should be able to understand directly the input of the user and take into account the context of the proof. The user should not have to specify everything to the lowest level in order to be useful. That is, a user should be able to use tactics and concepts much higher than simple predicate calculus. The feedback in case of error should highlight which element of the proof is problematic. This verification should be interactive. This means the user should be able to query the system about the validity of a proof currently in progress. This would ease the writing of

the proof as the user would be able to check line by line if the input is accepted.

Chapter 3 shows an implementation that aims at being a proof of concept of such ideal system. It tries to include some of the discussed features.

2.4 Verification delegation to Lean

This section describes the main strategic choice of this work. That is, the proof verification is delegated to a specialized language / software. It was first considered to implement a simple verification system considering a small subset of mathematics. Constructing a verification system for example for first order logic seemed at first feasible. Such systems exist without proof saving in simple format for example at the link :

<https://github.com/OpenLogicProject/fitch-checker>

However, in the goal of this proof of concept, more advanced and practical examples were desired, real number analysis for example. As such, after some tests, even for a very restricted set of mathematics it became clear that building a proof assistant from scratch would require much work not completely related to the initial goal of interacting by natural language. There are many possible directions to build a new theorem prover as shown in the general review of [4]. The decision was then taken to use an external language / software to handle the proof and to focus on an interface from a user and that external system (schematically shown at Figure 2.4).

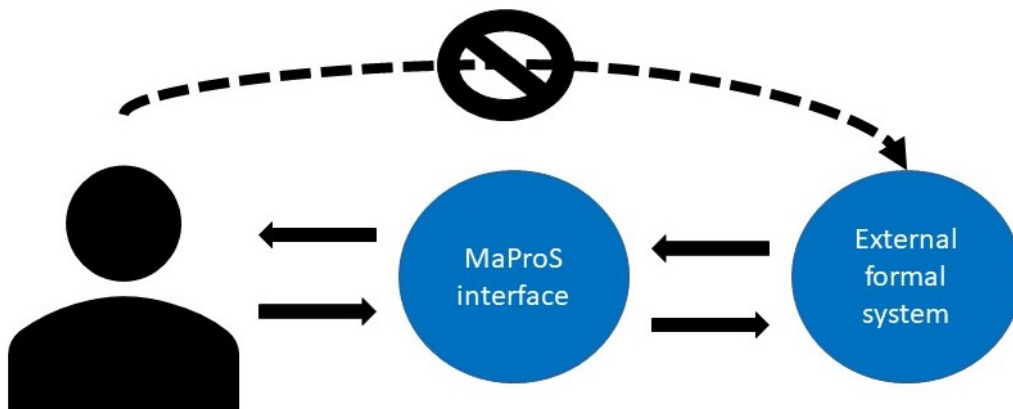


Figure 2.4: Schematic interaction with external system

The focus was placed on how to ease the interaction between the user and this external system. That is, to add an interface such that the user has a more

friendly experience and does not need to directly communicate with the external more technical system.

The current proof of concept uses the language Lean to perform proof construction and verification. The following highlights some aspects of Lean to understand how it is used.

Lean is a theorem prover developed by Microsoft¹. There is a large and very active community that aims at formalizing most mathematics in Lean². The community has published several papers describing design, goals and implementation of the library such as [7, 9]. The current stable version of Lean is Lean 3 which was first released in 2017. This is the version that will be used in this thesis and it is also the version currently used by the community. However, Lean 4 is in active development.

2.4.1 Dependant type theory

Lean is based on a dependant type theory called calculus of construction. The main idea is that every mathematical object is an element of a particular set. This means a theorem is also an element of a specific set. Declaring a theorem is to specify to what set it belongs and proving the theorem is to prove that the set to which it belongs is non-empty (for example by constructing an element). Usually a theorem belongs to a set of functions, that take as arguments the hypotheses and return the result. For example, Figure 2.5 shows the type of a theorem specifying that the limit of the sum of two sequences is the sum of the two respective limits.

```
sum_limit : ∀ (a b : ℕ → ℝ) (l_a l_b : ℝ), is_limit a l_a → is_limit b l_b → is_limit (a + b) (l_a + l_b)
```

Figure 2.5: Type of a theorem indicating that the sum of two series converges to the sum of the respective limits of each series

As seen, *sum_limit* is a function that take as parameters two real sequences ($\mathbb{N} \rightarrow \mathbb{R}$), two real numbers (*l_a* and *l_b*), one element of type *is_limit a l_a*, one element of type *is_limit b l_b* to finally get an element of type *is_limit (a+b) (l_a + l_b)*. The proof of this theorem is indicated at Figure 2.6.

¹Main website is available at <https://leanprover.github.io/>

²Website of the community is available at <https://leanprover-community.github.io/index.html>

```

definition is_limit (a :  $\mathbb{N} \rightarrow \mathbb{R}$ ) (l :  $\mathbb{R}$ ) :=
 $\forall \epsilon > 0, \exists N, \forall n \geq N, |a\ n - l| < \epsilon$ 
theorem sum_limit
  (a b :  $\mathbb{N} \rightarrow \mathbb{R}$ )
  (l_a l_b :  $\mathbb{R}$ )
  (ha : is_limit a l_a)
  (hb : is_limit b l_b)
  :
  is_limit (a+b) (l_a+l_b)
  :=
begin
  intros  $\epsilon$  H $\epsilon$ ,
  have H2 :  $\epsilon / 3 > 0$  := by obvious_ineq,
  cases ha ( $\epsilon / 3$ ) H2 with Na Ha,
  cases hb ( $\epsilon / 3$ ) H2 with Nb Hb,
  use Na+Nb,
  intros n Hn,
  have Hn1 :  $n \geq Na$  := by obvious_ineq,
  have Hn2 :  $n \geq Nb$  := by obvious_ineq,
  have H3 := Ha n Hn1,
  have H4 := Hb n Hn2,
  rw abs_sub_lt_iff at H3 H4 ⊢,
  split;
  have H5 : (a+b) n = a n + b n := pi.add_apply a b n;
  obvious_ineq,
end

```

Hypotheses

Result type after applying hypotheses

Proof to construct an element of the set
is_limit (a+b) (l_a + l_b)

Figure 2.6: sum_limit theorem

As seen, after given an identifier to the theorem (*sum_limit*) one declares a context, that is, say hypotheses to use for the theorem. Then one indicates the resulting type of what should give the theorem after applying all hypotheses. Followed by the "=" is the proof on itself that in this case is a sequence of tactics to solve the goal. A tactic is a specific instruction to either produce new sentences or to change the current goal. For example, the tactic *have* in this example :

```
have H2 :  $\epsilon / 3 > 0$  := by obvious_ineq,
```

creates a new variable in the context named H2 without changing the goal. On the other hand, the tactic *use* in this example :

```
use Na + Nb,
```

changes the current goal which is of the kind $\exists N$ by specifying that $Na+Nb$ will serve as N for this goal. Tactics are not the only way to construct proofs but they are extremely useful. One reason for that is that Lean allows the user to write its

own tactics. This kind of meta-programming grants users ways to construct proofs more easily and with less steps. A few other examples of tactics are :

- *intro* : if the goal is of the type $\forall x : y$ or $x \Rightarrow y$ then adds in the local context x as an hypothesis and changes the goal to y .
- *simp*³ : high-level tactic that tries to close the goal with the current context and several lemmas, theorems.
- *tauto* : high-level tactic that tries to close logical goals. For example a goal of type $A \wedge B \Leftrightarrow C$ should be solved by this tactic with the correct hypotheses.

2.4.2 Reasons for choosing Lean

Lean is in active development as a new version Lean 4 is currently in progress. It thus uses modern concepts (extensive use of Unicode, experience with calculus of construction). However, the main choice that drove the use of Lean is the community. Indeed, it is very large and well organized. They have created a huge library called *mathlib*. The library overview is available at this link :

<https://leanprover-community.github.io/mathlib-overview.html>

At the time of writing, the library contains 25586 definitions and 56270 theorems written by 167 contributors⁴. It is in active development and pull requests are welcome. In this library, concepts are clearly structured and defined from the bottom. There are also many useful tactics.

2.5 Other related projects

This section discusses recent works found that are related to the implementation presented in chapter 3.

2.5.1 Formal Abstracts

This project is available at

- Description : <https://formalabstracts.github.io/>
- Source code : <https://github.com/formalabstracts/formalabstracts>

³More information about this tactic at <https://leanprover-community.github.io/extras/simp.html>.

⁴Statistics are available at https://leanprover-community.github.io/mathlib_stats.html

This project is described as *A formal abstract, or fabstract for short, is a formalization of the main results (constructions, definitions, proofs, conjectures) of a piece of informal mathematics, such as a research paper*. It resembles this thesis as it involves comprehension and processing of mathematics written in a close to natural language (such as in a research paper). It seems the project also intends to use Lean extensively. The project is currently said to be in design phase. The GitHub repository is inactive since 2019. It counts for the moment 16 contributors. By peeking at the repository, it seems that they do not have yet functional features.

2.5.2 HolPy

The source code for this project is available at

`https://github.com/bzhan/holpy`

The description of the work is in the paper [13]. It is described in its title as a *Interactive Theorem Proving in Python*. It has some similarities with this work as the project is also a web application. The specific stack of technologies is different as the backend is written in Flask and the frontend with the framework Vue.js. It also aims as being rather user friendly. It does not however seem to have the same focus on translating natural language to formalized mathematics. Some of the ideas seem to be inspired from Lean while innovating on other aspects.

2.5.3 Lean formatter

This project is available at

`https://github.com/leanprover-community/format\_lean`

It allows conversion of a Lean file to html. By annotating in comments in the code Latex, one can build a sort of interactive proof. An example is available at

`https://leanprover-community.github.io/format\_lean/example/sandwich.html`

This example served as basis for the current implementation for basic instructions in Lean and the sandwich theorem example. This project is similarly to this thesis, concerned with connecting Lean instructions with equivalent natural language sentences. The translation however is made by the user and not automated.

Chapter 3

Implementation

This chapter describes the current implementation. It is a proof of concept of the discussed ideal system. To grant higher accessibility and to exploit modern technologies, the designed system is a website application. Source codes for both frontend and backend server can be found respectively at :

- backend : <https://github.com/azarzadavila/mapros>
- frontend : <https://github.com/azarzadavila/mapros-frontend>

Technology stack The backend server is written with Python with the framework Django while the frontend is written in javascript with the framework React. My own skills in these technologies were an important factor of decision. A more detailed list of the main used libraries would be the following :

- **Python**
As the task seemed to involve manipulation of strings, parsing, natural language processing Python seemed an excellent choice due to the presence of numerous libraries in these fields. Performance was not currently a first priority. Switching to a low-level programming language such as C was not considered.
- **Django**¹
This backend framework is a python web framework. It allows to build robust and quick applications. It is modern and a popular choice.
- **Django Rest Framework**²
It combines with Django to provide a rest API to allow another framework to handle the frontend.

¹Homepage at <https://www.djangoproject.com/>.

²Homepage at <https://www.django-rest-framework.org/>.

- **React**³
Frontend javascript framework developed by Facebook. It allows to structure the frontend code and guides its coding style.
- **React Bootstrap**⁴
Used with React to provide good looking UI without much hassle.
- **MathQuill**⁵
Web mathematical formula editor integrated in the frontend. In particular, a wrapper available at this link

`https://github.com/viktorstrate/react-mathquill`

was used to integrate it with React.

This list is not exhaustive and can be completed by the *requirements.txt* available at the repository.

Installation Installation guides can be found at :

- backend : `https://github.com/azarzadavila/mapros/blob/main/installation.md`
- frontend : `https://github.com/azarzadavila/mapros-frontend/blob/main/installation.md`

Installation requires installing Python with the specified dependencies for the backend. One must also create a local database to connect with the server. The frontend mainly requires Node and npm.

Lean The system also requires a version of Lean with *mathlib*. For this reason, it is not very straightforward to include the implementation on a free hosting service. Guide for installing Lean can be found at `https://leanprover-community.github.io/get\_started.html`.

3.1 Overview

The following section gives an overview of the current system features. Appendix A shows some additional views of the current implementation as well as one example of a complete proof.

³Homepage at `https://reactjs.org/`.

⁴Homepage at `https://react-bootstrap.github.io/`.

⁵Homepage at `http://mathquill.com/`.

Educational purposes One interesting use of collaborative system could be for educational purposes. With some of the discussed features of the ideal system it would be possible to build a platform for generating and solving proofs as exercises. For example, with some known theorems in an undergraduate course, teachers would like their students to prove a new theorem as an exercise. The teachers write the statement with the required hypotheses and send it to their students so they can write the proof and check its correctness with the system. The current system implements this kind of idea. Let us now dive in the concrete result. An overview video is available at

<https://youtu.be/xhAeL3VtkpU>

It is advised that the reader watches the video before continuing in order to visualize the current implementation.

3.1.1 Login/Create account

In order to send and receive proofs one needs an account in order to associate the proof to a user. The current system has a very simple identification system based on token authentication. Figures 3.1, 3.2 show some of the steps. It is also possible to access some of the features without being logged in.

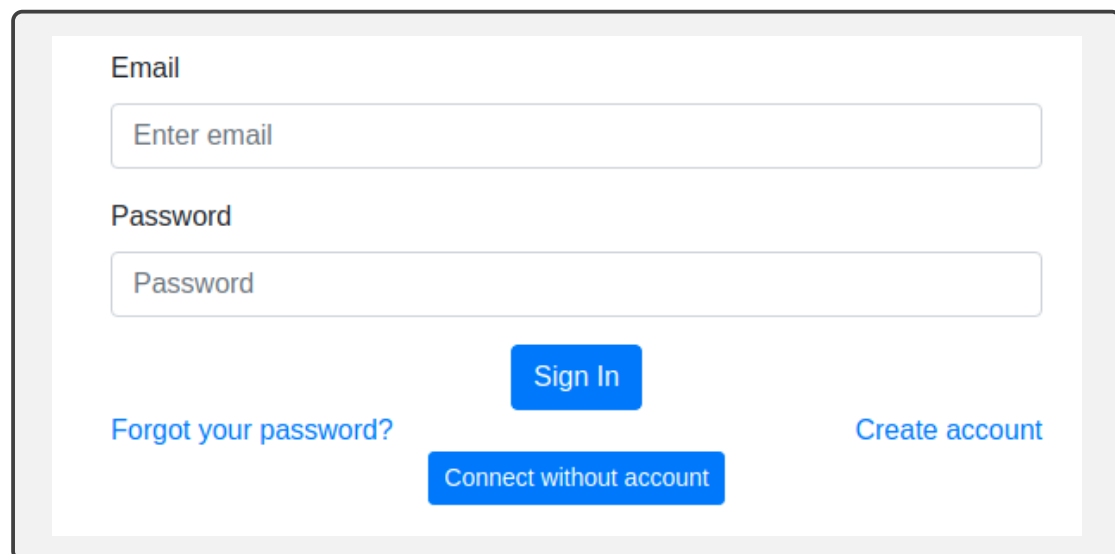
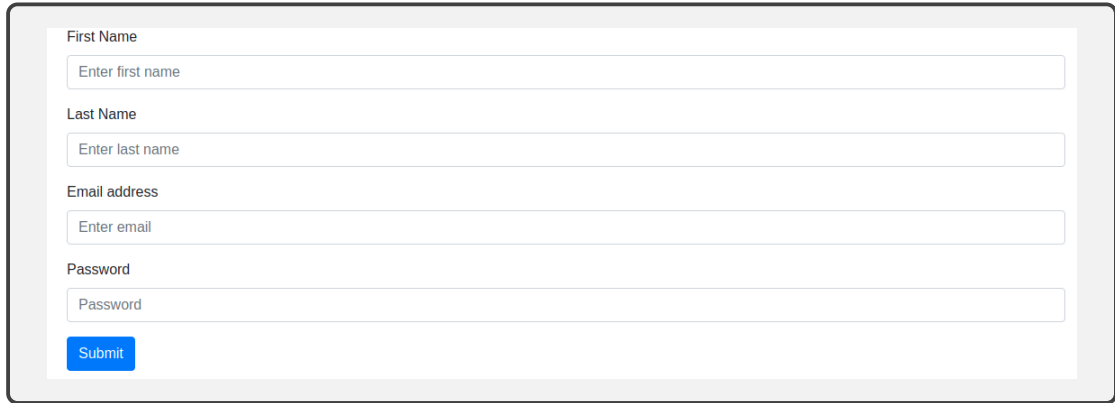
The image shows a login interface within a light gray rounded rectangle. It contains two input fields: 'Email' with a placeholder 'Enter email' and 'Password' with a placeholder 'Password'. Below the password field is a blue 'Sign In' button. At the bottom, there are three links: 'Forgot your password?' on the left, 'Connect without account' in a blue button in the center, and 'Create account' on the right.

Figure 3.1: Login view



A registration form titled "Create an account" with four input fields: "First Name", "Last Name", "Email address", and "Password". Each field has a placeholder text: "Enter first name", "Enter last name", "Enter email", and "Password". A blue "Submit" button is located at the bottom left of the form.

Figure 3.2: Create an account

Figure 3.3 shows the menu once a user is logged in.

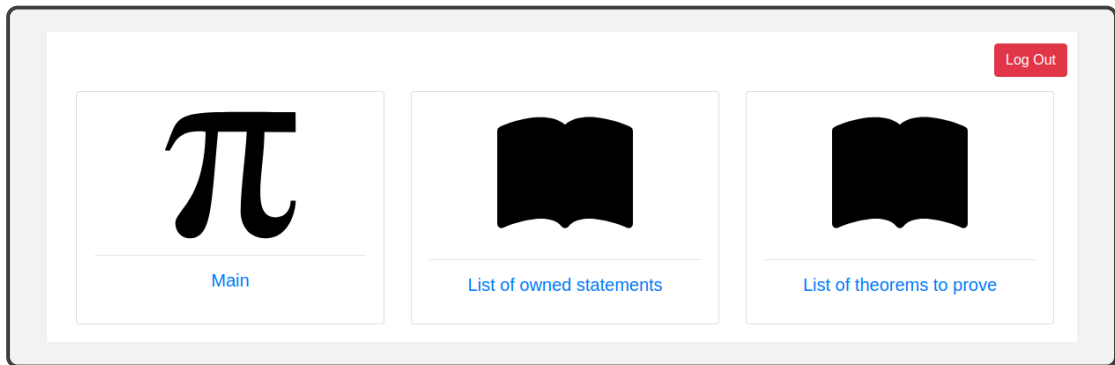


Figure 3.3: Menu logged in

Three different aspects are in this menu. Let us review them one by one.

3.1.2 Write a proof

Figure 3.4 shows the interface to write a complete proof. This is the interface one arrives when clicking on *Main* in the menu.

HOME

Theorem Name : sandwich

If, +

a_n, b_n, c_n are real-valued sequences -

$l \in \mathbb{R}$ -

H1 $a_n \rightarrow l$ - 1 Hypotheses

H2 $c_n \rightarrow l$ -

H3 $\forall n : a_n \leq b_n$ -

H4 $\forall n : b_n \leq c_n$ -

Then, $b_n \rightarrow l$ S 2 Goal $b_n \rightarrow l$

Proof: Let ϵ - S Lean $\exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow |b n - l| < \epsilon$

3 Proof

A1 $\epsilon > 0$

Proof in progress

+

Figure 3.4: Interface to write a complete proof

This feature is accessible by both authenticated and anonymous users as it does not require any data storage. It allows to write a theorem statement that is composed of hypotheses and a goal along with its proof. It is interactive and can be separated in different parts. The first part is the name of the theorem. Then one can see the ability to add, modify and delete hypotheses indicated in Figure 3.4 as number 1. Then there is the *goal* to write, noted as number 2 in Figure 3.4, that is what the theorem actually states with the given hypotheses. Finally, the last part is the proof on itself.

Feedback Throughout views like this the user can ask for feedback from the system. This feedback and additional information are visible with the following color codes

- Gray banners either represent sentences generated by the systems or information available in read-only format.

- Blue banners indicate progress without error. It is used for either the current goal state or to indicate as a general feedback that the proof is in progress.
- Red banners are used for errors. They can be accompanied by a small label indicating the type of error.
- Small black labels are used to indicate identifiers of generated sentences.
- Small yellow label where *Lean* is written is used as a warning indicating that the message could not be correctly translated which means that the output is in Lean without being translated.

Math Input

Inputs in Figure 3.4 are not simple text input (with the exception of the theorem name). These type of inputs will be denoted in the following as math inputs. They actually are a combination of text inputs with MathQuill inputs. Specifically, by pressing *CTRL+Enter* while being in the field a new MathQuill input is created as shown in Figure 3.5.

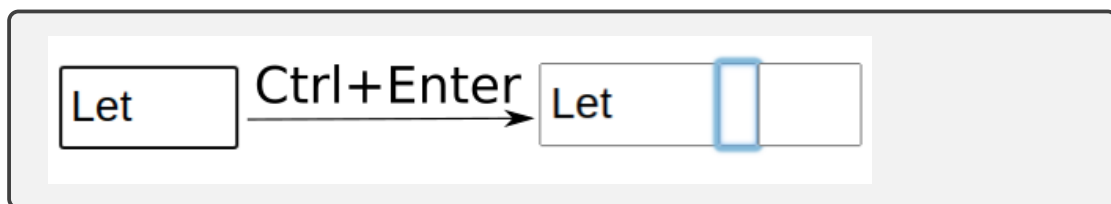


Figure 3.5: Create a MathQuill input

In this figure the left- and right-most inputs are simple text fields while the one in the middle allows to write mathematical formulas. In that field, Latex instructions are immediately translated as shown in Figure 3.6 giving the user a fluid experience.



Figure 3.6: MathQuill changing `\epsilon` to its symbol

Including this seemed a much better experience than only permitting writing static Latex instructions. The integration of MathQuill was unexpectedly quite a

hassle. The main reason for it is that its integration with React is not very stable through the used wrapper and sometimes created unexpected changes.

Hypotheses

One can add or remove hypotheses by clicking respectively on buttons indicated at Figure 3.7.

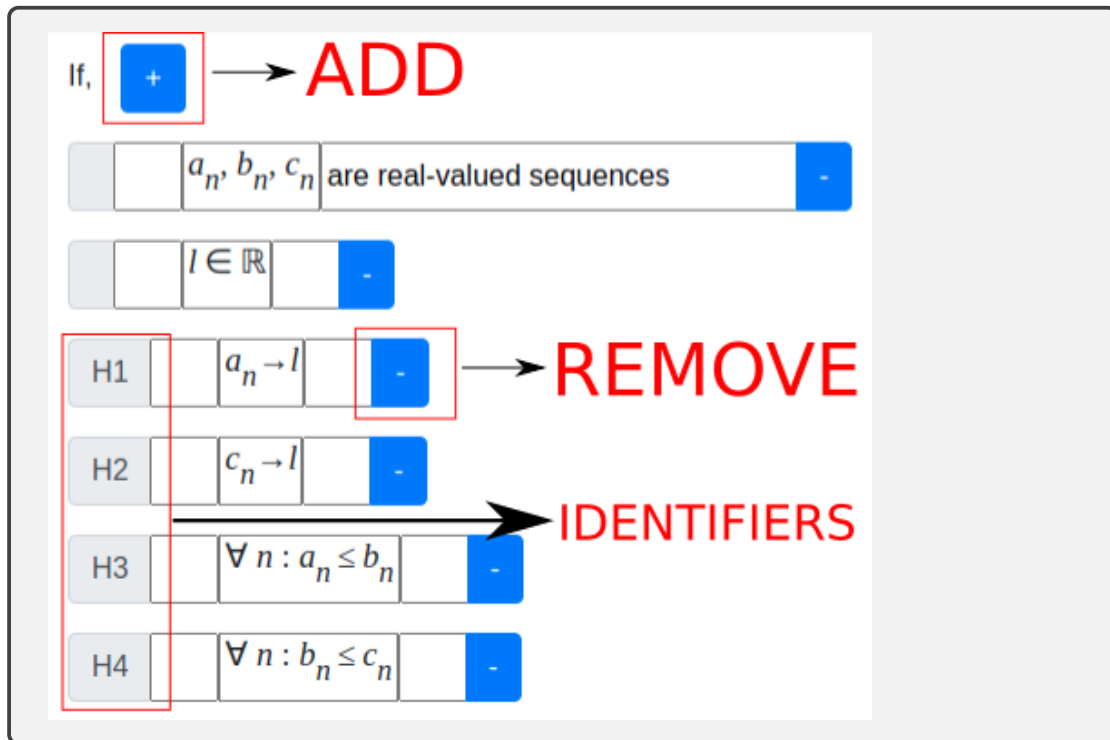


Figure 3.7: Hypotheses manipulation

Each hypothesis line is made up of an identifier label and the hypothesis itself as a math input. The identifier is not chosen by the user but generated automatically by the system. Some hypotheses have anonymous identifiers. That is the case for example for variable declarations such as seen in Figure 3.7 : $l \in \mathbb{R}$. One reason for this is that in a proof if one wants to invoke the declaration or a property of the variable, one would write something like *by definition of l* . Another reason is that in Lean declaring an hypothesis or a new variable is very similar and the name of the variable would be equivalent to the name of the hypothesis.

Goal

The goal represents what the theorem actually states. The goal is written in a math field. It can use any hypothesis and contain quantifiers. A user can ask at this point feedback from the server by clicking on the state button shown in Figure 3.8.



Figure 3.8: Ask State at initial goal

The response should indicate on the right the same goal and feedback should contain no error except the fact that the goal is not yet proven. Two types of errors can arise. The goal or one of the hypotheses written in natural language might not have been understood by the system as shown in Figure 3.9.

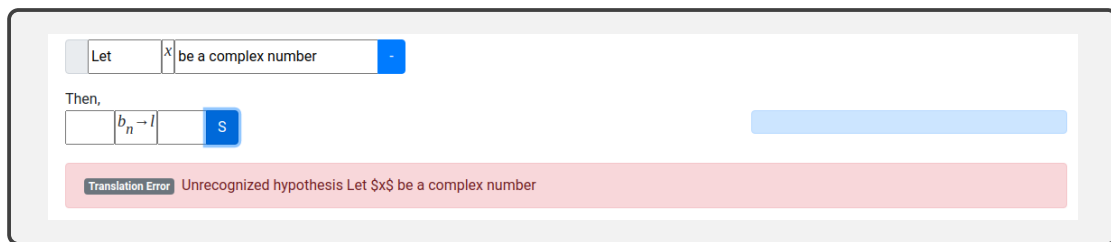


Figure 3.9: Unrecognized hypothesis error - Failed to translate to Lean

The other type of error is produced by Lean as shown in Figure 3.10.

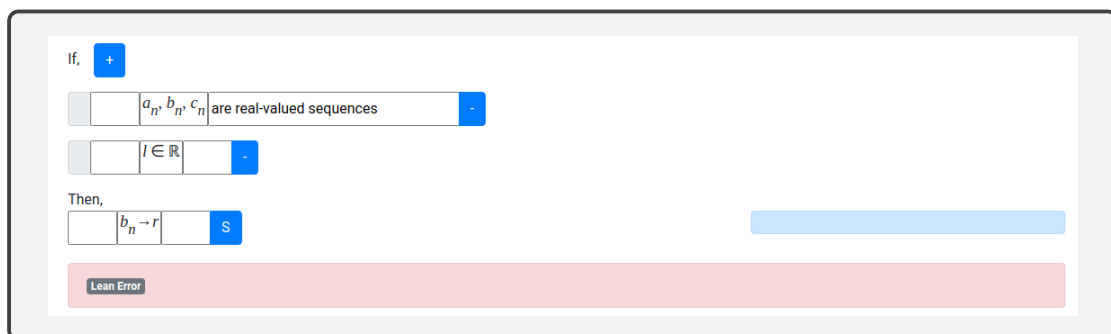


Figure 3.10: Lean feedback indicates an error - r has not been declared

Proof

Proof lines can be added in the same way as with hypotheses by clicking respectively on buttons indicated at Figure 3.11.

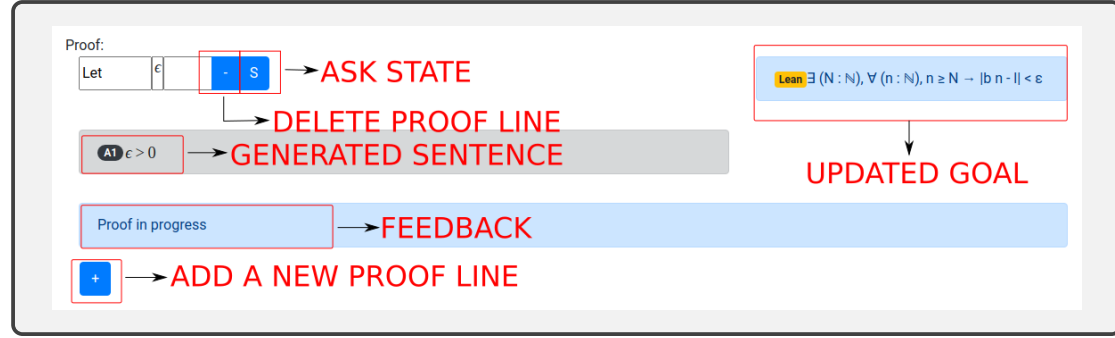


Figure 3.11: Proof line

A proof line can be seen as an instruction to make progress in the proof such as the specification of variable in a quantifier, or the construction of a new variable. Each proof line is composed of a math input and button to ask for the state of the proof at that point as seen in Figure 3.11. Asking for the state can have several effects. In case of error, the two same types of errors can arise as previously. A proof line written in natural language might not have been understood or an error has happened in Lean. In case of normal execution, the updated goal state should appear. A proof line can also have as effect to generate some new sentences as shown in Figure 3.11. These sentences are given identifiers automatically. These sentences are generated from Lean. If a failure occurs while trying to translate the Lean output, the system chooses to output the raw Lean feedback as a default fallback. This is seen by a warning in Figure 3.11 in the goal. These sentences can then be referenced by their identifiers in new proof lines for example in Figure 3.11 by (A1). At the end of the proof, the feedback indicates the goal is achieved as shown in Figure 3.12.

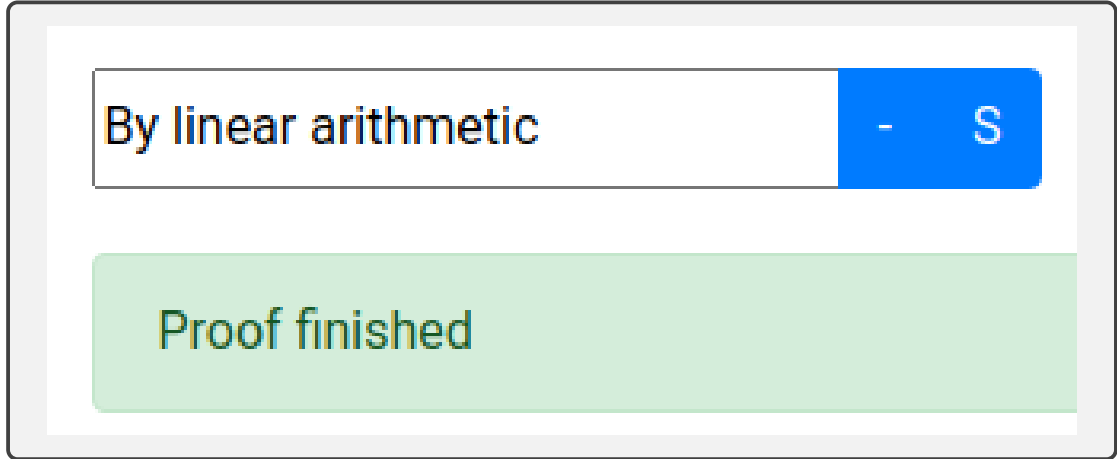


Figure 3.12: Feedback indicating success of proof

3.1.3 Write a statement and send it

Let us now consider the case of a user that would want to write only the statement of a theorem with its hypotheses to send it to other users. That would be the example of teacher that would write exercises for their students. This feature is available in the menu through the *List of owned statements* button. Figure 3.13 shows the view of all written statements by the user. In this view a user can add a new statement delete one or manage an existing theorem.



Figure 3.13: List of owned theorem statements

By clicking on adding a new statement or by selecting an existing one, the user arrives at an interface similar as shown in Figure 3.14.

BACK

Theorem Name : sandwich

Save

Send

Manage

If,

+

a_n, b_n, c_n are real-valued sequences

-

$l \in \mathbb{R}$

-

H1

$a_n \rightarrow l$

-

H2

$c_n \rightarrow l$

-

H3

$\forall n : a_n \leq b_n$

-

H4

$\forall n : b_n \leq c_n$

-

Then,

$b_n \rightarrow l$

S

$b_n \rightarrow l$

Proof in progress

Figure 3.14: Edition of a statement

This interface is very similar to the one to write a complete theorem proof. In this view however, there is no way to add proof lines as it is only the proof statement. There is on the other hand a button to save a new or updated statement. There is also a *send* button that shows the view at Figure 3.15 allowing the user to send the statement to selected users.

List of users	
User3 USER3	☒
User1 USER1	☐
User2 USER2	☒

Cancel Send

Figure 3.15: Send statement to users 2 and 3

The *manage* button shows another view that allows the user to delete some of the receivers as shown in Figure 3.16.

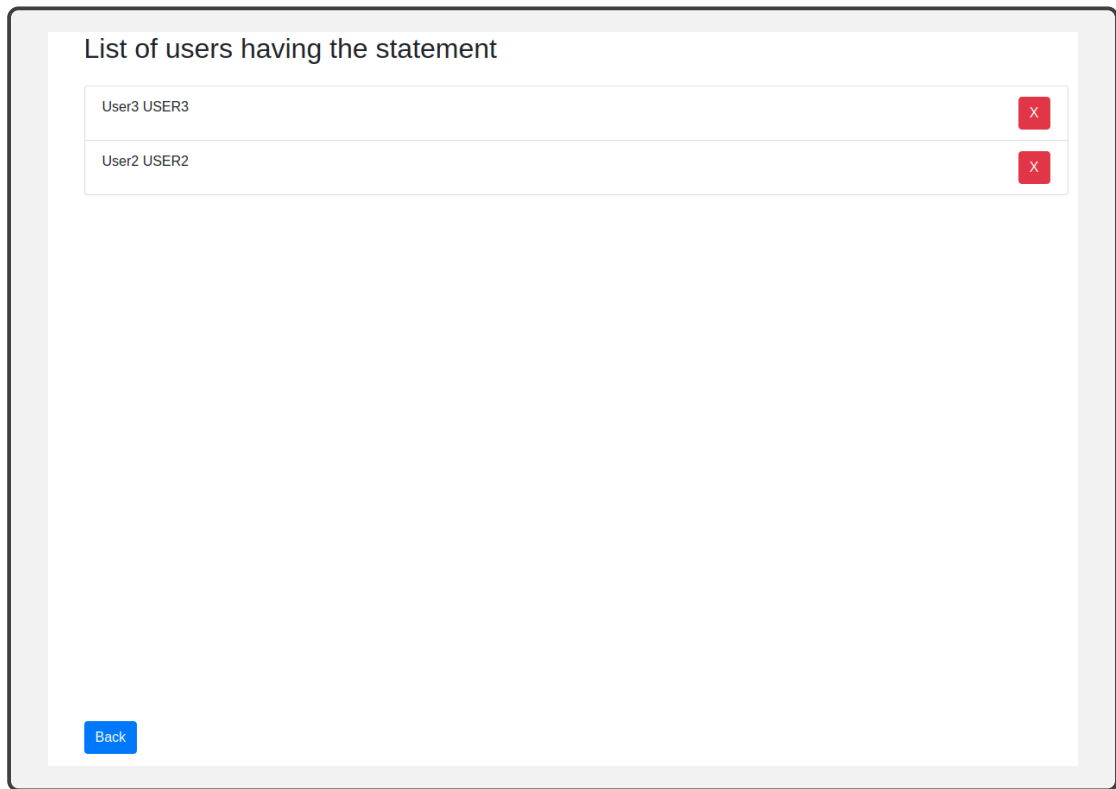


Figure 3.16: Remove some users from the list that received the statement

3.1.4 Write a proof for a sent statement

A user can see the statements that were sent to him/her. This feature is available in the menu through the *List of theorems to prove* button. Figure 3.17 shows the list of sent theorems.



Figure 3.17: List of statements sent to a user

By clicking on a specific statement, one arrives at Figure 3.18.

BACK

Theorem Name : sandwich

Save

If,

a_n, b_n, c_n are real-valued sequences

$l \in \mathbb{R}$

H1

 $a_n \rightarrow l$

H2

 $c_n \rightarrow l$

H3

 $\forall n : a_n \leq b_n$

H4

 $\forall n : b_n \leq c_n$

Then,

$b_n \rightarrow l$

S

$b_n \rightarrow l$

Proof in progress

Proof:

S

+

Figure 3.18: Prove a theorem with fixed statement

Again, this interface behaves similar to the interface to write a complete proof. However, the hypotheses and goal are read-only. One can save the current state of the proof. This allows the user to start writing the proof and save it even if it is not yet complete.

3.2 System structure

The system structure is mainly separated in three elements :

- Lean : verifies and builds a proof
- Backend python server : translation between Lean language and natural language + database interaction
- Frontend react server : user interface and some minor language preprocessing

Figure 3.19 shows a schematic model of the interactions.

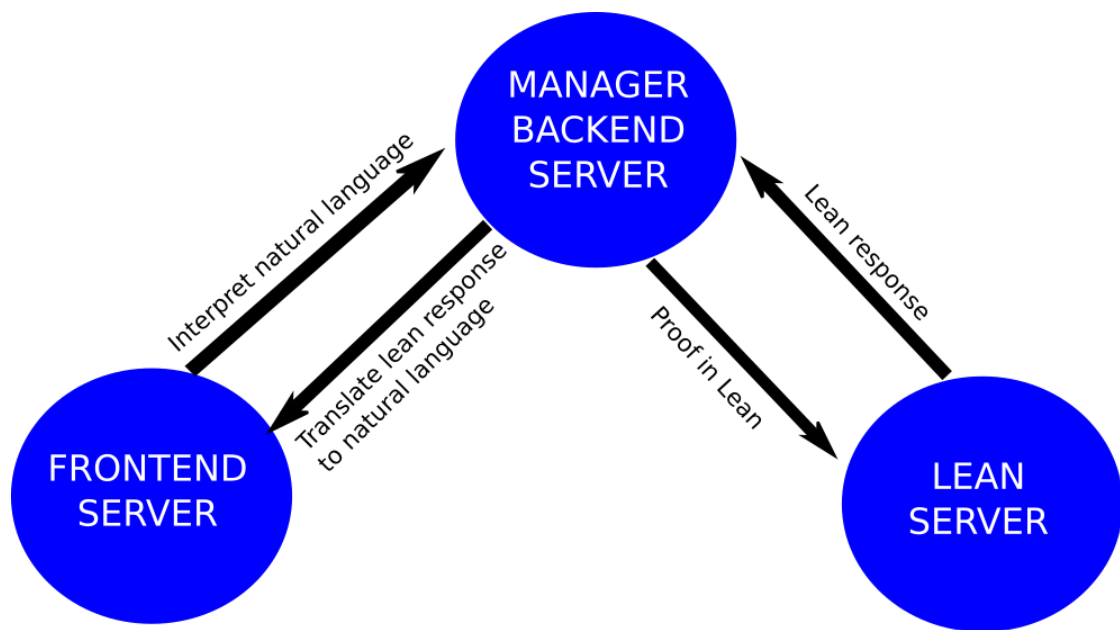


Figure 3.19: Translation structure

3.2.1 Database

Figure 3.20 shows a simplified version of the SQL database.

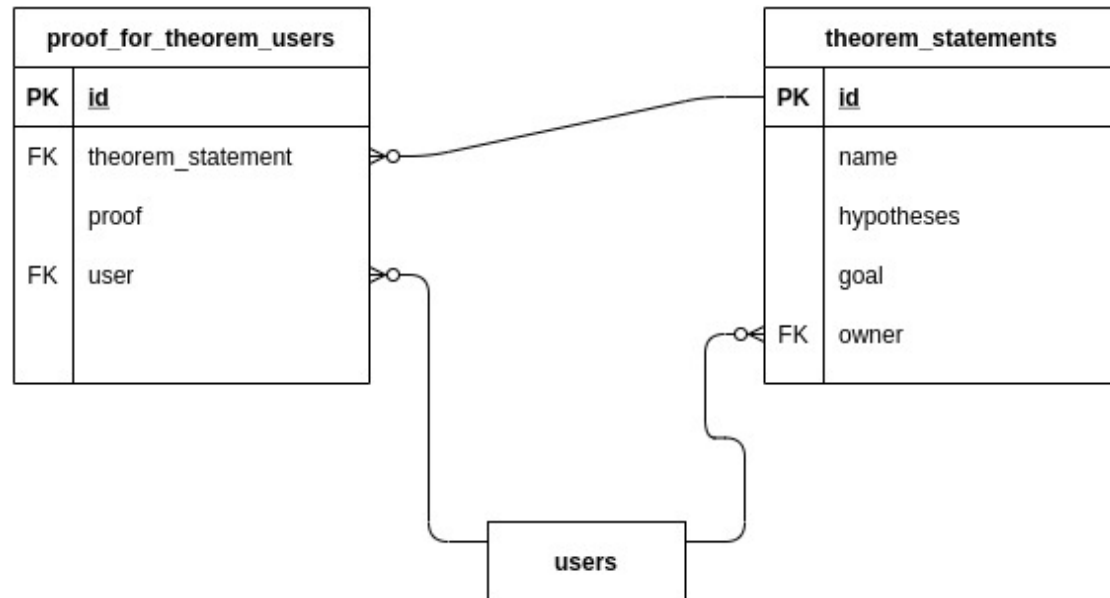


Figure 3.20: Database diagram

The current implementation uses PostgreSQL. However, all SQL instructions are handled by the interface of Django. The specifications for the models that create the tables are in the file *models.py* :

<https://github.com/azarzadavila/mapros/blob/main/main/models.py>

The different visible elements are directly written to the database. User table is handled with defaults by Django. This includes safe storing of password by hashing, strength verification, email format and other standard aspects. The hypotheses and proofs are written closely to the input of the user in a single string. The hypotheses and line proofs are separated with a line break. The database does not store information that would be retrieved by compilation with Lean. As such, Figure 3.21 shows an example of storing some proof lines.

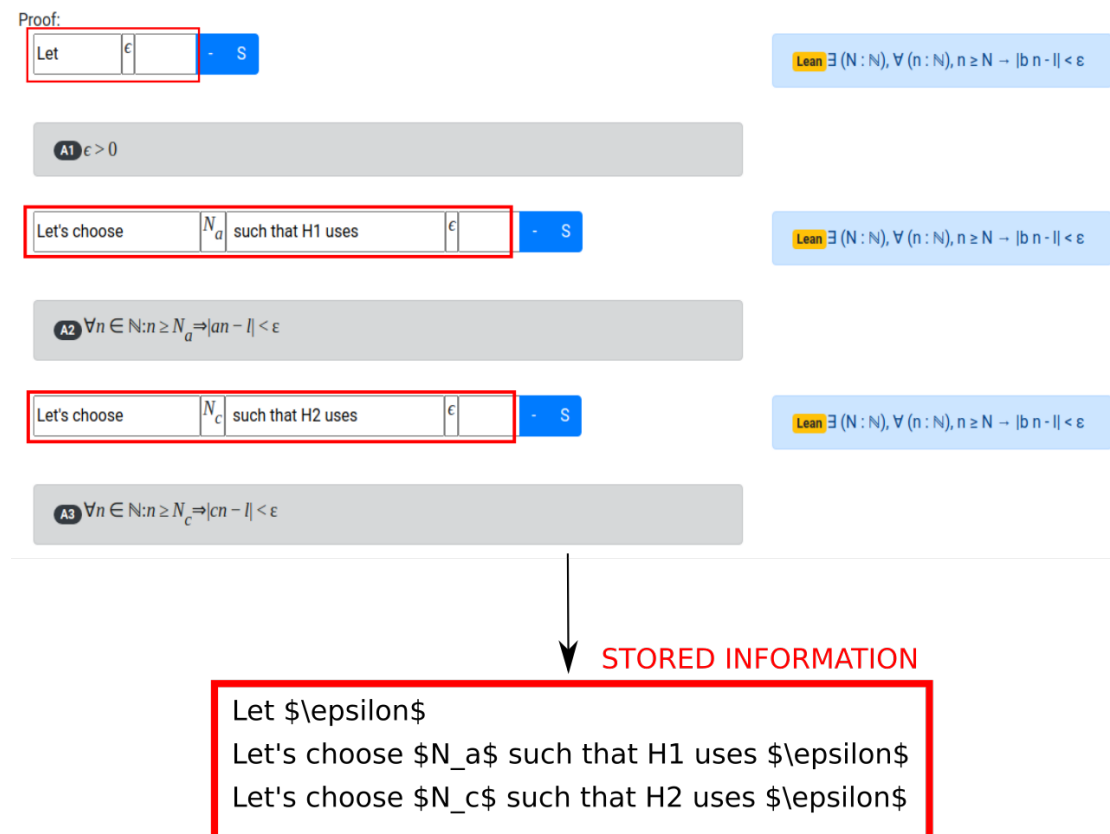


Figure 3.21: Proof lines stored in database

In this example, the sentences generated by Lean (A1, A2, A3) and feedback are not stored. The reason not to store this information is that it can be recompiled at any time.

3.3 Translation

The translation part seen in the previous section is a very difficult task. There is here a crucial assumption that at each hypothesis, goal or proof line, one can classify the instruction in a set of finite proof instructions. There are several ways to deal with this task some of them will be discussed in 4. At first, I considered using classical parser algorithms such as LALR(1) and Earley. Great libraries exist in Python for this purpose such as Lark⁶. However, even if these algorithms are very efficient, after trying it, I found it difficult to use it to this use case. The reason is that the considered translations can sometimes be very contextual. That

⁶Project available at <https://github.com/lark-parser/lark>

is, the translation of a given tactic in natural language might be different depending on the goal. As such, for now the system considers that each element that has to be understood in natural language or Lean implements the following interface in Figure 3.22.

```
class Language:
    def to_lean(self) -> str:
        raise NotImplementedError

    def to_natural(self, in_math=False) -> str:
        raise NotImplementedError

    @classmethod
    def from_natural(cls, s: str, context=None, in_math=False):
        raise NotImplementedError

    @classmethod
    def from_lean(cls, s: str, context=None):
        raise NotImplementedError
```

Figure 3.22: Language interface

This makes the bridge between natural language and Lean by first creating a Python object that matches the concept. Methods, *to_natural* and *to_lean*, respectively output the object as a string in natural language and Lean. On the other hand, methods, *from_natural* and *from_lean*, generate the Python object from a string respectively in natural language or in Lean. Two abstract classes implement this interface : *Sentence* and *Tactic*. The class *Sentence* corresponds to a mathematical expression while the class *Tactic* tries to match a proof instruction. The class *Tactic* also adds another method *to_extract* that retrieves what expressions are to be retrieved in the Lean output. This is the case for the sentences that are produced automatically in Lean as seen in Figure 3.11. It is interesting to note that due to the use of MathQuill the natural language contains Latex math mode while Lean uses Unicode. This sometimes causes a bit of a hassle. For example, special character ϵ is directly written in Lean (as it exists in Unicode) while it is written `$\epsilon$` in natural input.

3.3.1 Context

As mentioned, translation of a given line might depend on the current context, which means on the knowledge that has been accumulated so far on each element. For example, let us consider the following sentence in Lean :

`a i < M`

How should it be translated in natural language ? Well depending if **a** is considered as a function or as a sequence it could be either $a(i)$ or a_i . A similar example can occur from natural language to Lean. Let us consider the sentence :

We claim a_i works

How should this be translated in Lean ? Depending if **a** is considered as a sequence or not, it could be either *use (a i)* or *use a_i*, the latter meaning a_i is an identifier and not a point of a sequence. For these reasons, the methods shown at Figure 3.22 take as an argument a context that might inform on such questions.

3.3.2 Parsing approach

As mentioned above the current method does not use a specific parser algorithm. It does however use extensively regular expressions. Given a tactic in natural language, an instance of *Manager* checks sequentially in a given order each tactic until one is correctly matched or none is. Each tactic checks if it is being matched by checking if the line first matches a specific regular expression. After that, the tactic might require some parts to match other sentences or tactics making it a compound system. For example, the tactic *ByInequalityProperties* indicated at the following code allows the user to make a statement to be proved by inequality properties.

```

1  def _sentences_match_byinequalityproperties():
2      return [Inequality]
3
4
5  class ByInequalityProperties(Tactic):
6      def __init__(self, ident, sentence):
7          self.ident = ident
8          self.sentence = sentence
9
10     def to_lean(self) -> str:
11         return (
12             "have "
13             + self.ident
14             + " : "
15             + self.sentence.to_lean()
16             + " := by obvious_ineq"
17         )
18
19     def to_natural(self, in_math=False) -> str:

```

```

20         return "By inequality properties, " + self.sentence.to_natural()
21
22     @classmethod
23     def from_natural(cls, s: str, context=None, in_math=False):
24         match = re.fullmatch(r"By inequality properties, (.+)", s)
25         if not match:
26             return None
27         sentence = from_natural(
28             match[1], context, _sentences_match_byinequalityproperties(), in_math
29         )
30         if not sentence:
31             return None
32         ident = context.next_anonymous()
33         return cls(ident, sentence)
34
35     @classmethod
36     def from_lean(cls, s: str, context=None):
37         match = re.fullmatch(r"have (\w+) : (.+) := by obvious_ineq", s)
38         if not match:
39             return None
40         sentence = from_lean(
41             match[2], context, _sentences_match_byinequalityproperties()
42         )
43         if not sentence:
44             return None
45         return cls(match[1], sentence)
46
47     def to_extract(self) -> List:
48         return [self.ident]

```

As seen in *from_natural*, after the tactic matches a regular expression it checks if a specific part matches a sentence of type *Inequality*. Figure 3.23 shows an example of this applied to a concrete proof.

Let	$N = \max(N_a, N_c)$	- S	Lean $\exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow \ b_n - l\ < \epsilon$
Lean $N := \max N_a N_c$			
We claim	N works	- S	$\forall n \in \mathbb{N} : n \geq N \Rightarrow \ b_n - l\ < \epsilon$
By inequality properties,	$N_a \leq N$	- S	$\forall n \in \mathbb{N} : n \geq N \Rightarrow \ b_n - l\ < \epsilon$
A4 $N_a \leq N$			

Figure 3.23: By inequality properties - example

Matching in class *Sentence* and matching Lean output follows the same principle.

3.3.3 List of current possible translations

This subsection shows a summary of the possible current translations. This list is also available on the repository at :

https://github.com/azarzadavila/mapros/blob/main/user_guide.md

Hypotheses The current hypotheses accepted are :

- Declaring real-valued sequences.
For example, ' a_n, b_n, c_n are real-valued sequences'.
- Real number declaration.
For example, ' $r \in \mathbb{R}$ '.
- Declaring the limit of a sequence.
For example, ' $a_n \rightarrow r$ '.
- For all type of declaration. Currently, it has to be followed with an inequality constraint.
For example, ' $\forall n : a_n \geq b_n$ '.

Tactics (to be written in proof lines) The current accepted tactics are :

- Do something on all subgoals combined with a split tactic. It will apply the next tactic to all subgoals.

For example, ‘Let’s split the goal and do on all subgoals.’

- *Let* when the current goal is to prove the limit of a sequence.

For example, ‘Let ϵ ’.

- Transform a previous sentence of sequence limit of type " $\forall \epsilon > 0, \exists N$ such that ...", by choosing the epsilon, giving an identifier to the N and adding the hypothesis that the given epsilon is positive.

For example, ‘Let’s choose N_a such that H2 uses $\frac{\epsilon}{3}$ (with A2)’ where A2 says that $\epsilon/3 > 0$.

- Transform a previous sentence of sequence limit of type " $\forall \epsilon > 0, \exists N$ such that ...", by choosing the epsilon and giving an identifier to the N . Compared with the previous tactic, no additional hypothesis is needed if the identifier for the selected epsilon is positive at its declaration.

For example, ‘Let’s choose N_a such that H2 uses ϵ_b ’.

- Declare a new real number being the maximum of two other real numbers.

For example, ‘Let $N = \max(N_a, N_b)$ ’.

- With a goal of the type " $\exists$ identifier such that ...", specify an already declared variable that accomplishes that goal.

For example, ‘We claim N_c works’.

- State an inequality that is supposed to be directly derived from previous sentences.

For example, ‘By inequality properties, $N \geq N_a$ ’.

- Let a natural number on a current goal that is of the type "for all natural numbers : ...".

For example, ‘Let n ’.

- State an inequality by modus ponens.

For example, ‘ $a \leq b$ by A8 with A7’.

- Specify a variable in a sentence of type " $\forall x :$ ".

For example, ‘Let’s choose n in A2’.

- Change some inequality sentences of type " $|a - b| < c$ " to " $a - b < c \wedge -a + b < c$ ".

For example, 'Let's use absolute value inequality property on A10 and on A12'.

- Separate a sentence of type "S1 and S2" to two sentences S1 and S2.

For example, 'Let's separate A4'.

- Split a current goal of type "S1 and S2".

For example, 'Let's split the goal'.

- Prove the current goal by using linear arithmetic with the help of all previous sentences.

For example, 'By linear arithmetic'.

- Modus ponens to generate a new sentence.

For example, 'By A10 with A12'.

- State a sentence by definition of addition for functions.

For example, ' $(a + b)_n = a_n + b_n$ by definition of addition for functions'.

- Prove the current goal by using inequality properties with all previous sentences.

For example, 'By inequality properties'.

3.3.4 Method for increasing translation ability

The approach taken for now to increase the number of allowed sentences and tactics is to consider existing proofs. That is, writing code that would allow to input a given proof. The assumption here is that by adding the ability to write a particular proof one will also add code that will allow to write part of another proof thus increasing the general translation ability. For the proof of concept of this thesis, I considered proofs closely related to the sandwich theorem for real-valued sequences. The system currently allows at least to prove the following theorems :

- sandwich theorem
- limit of the sum of two sequences is the sum of the respective limits of each sequence

The intersection of set needed by these two proofs is significant. Hopefully, writing other similar proofs should not require to write many unrelated tactics.

3.4 Sandwich theorem example

This section contains the functional example of a proof for the sandwich theorem. This proof is inspired from <https://www.ma.imperial.ac.uk/~buzzard/docs/lean/sandwich.html>.

Figure 3.24 contains all hypotheses needed. It starts by declaring three number sequences. Two of them, a and c converge to the same real number l . The sequence b is between these two sequences. The proof is started by introducing ϵ as the goal is to prove that a sequence converges to a real number.

The screenshot shows a Lean theorem prover interface with a blue header bar containing a "BACK" button. Below the header, the "Theorem Name" is set to "sandwich" with a "Save" button. The interface is divided into two main sections: "If," and "Then,".

If, section:

- A text box contains the statement: a_n, b_n, c_n are real-valued sequences.
- A text box contains the statement: $l \in \mathbb{R}$.
- A hypothesis box labeled "H1" contains the statement: $a_n \rightarrow l$.
- A hypothesis box labeled "H2" contains the statement: $c_n \rightarrow l$.
- A hypothesis box labeled "H3" contains the statement: $\forall n : a_n \leq b_n$.
- A hypothesis box labeled "H4" contains the statement: $\forall n : b_n \leq c_n$.

Then, section:

- A goal box contains the statement: $b_n \rightarrow l$. To its right, a blue box displays the Lean goal: $b_n \rightarrow l$.

Proof section:

- A "Proof:" label is followed by a "Let" box containing ϵ , a minus sign, and a "S" button.
- Below this, a hypothesis box labeled "A1" contains the statement: $\epsilon > 0$.
- To the right of the "Let" box, a blue box displays the Lean goal: $\exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow \|b_n - l\| < \epsilon$.

Figure 3.24: Sandwich Theorem - 1

In Figure 3.25, ϵ is used in hypotheses H1 and H2. Choosing ϵ in both these hypotheses gives a natural number such that the difference between the sequence and l is lower than ϵ . We give names to these natural numbers : N_a, N_b . Then we name N the maximum of N_a and N_b and say this solves the goal. Additionally, two rather obvious inequalities are stated. These steps should be improved to a more natural language in future versions.

Let's choose	N_a	such that H1 uses	ϵ	- S	Lean $\exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow b \ n - l < \epsilon$
A2 $\forall n \in \mathbb{N} : n \geq N_a \Rightarrow a \ n - l < \epsilon$					
Let's choose	N_c	such that H2 uses	ϵ	- S	Lean $\exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow b \ n - l < \epsilon$
A3 $\forall n \in \mathbb{N} : n \geq N_c \Rightarrow c \ n - l < \epsilon$					
Let	$N = \max(N_a, N_c)$		- S		Lean $\exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow b \ n - l < \epsilon$
N Lean $N := \max N_a N_c$					
We claim	N	works	- S		$\forall n \in \mathbb{N} : n \geq N \Rightarrow b \ n - l < \epsilon$
By inequality properties,	$N_a \leq N$		- S		$\forall n \in \mathbb{N} : n \geq N \Rightarrow b \ n - l < \epsilon$
A4 $N_a \leq N$					
By inequality properties,	$N_c \leq N$		- S		$\forall n \in \mathbb{N} : n \geq N \Rightarrow b \ n - l < \epsilon$
A5 $N_c \leq N$					

Figure 3.25: Sandwich Theorem - 2

Figure 3.26 shows next steps that start by the elimination of $\exists$ in the goal by applying a tactic *Let n*. Then we state some sentences that are directly derived from the fact that sequence b is always between a and c . It is also said that $N_a \leq n$ which comes from the fact that $N_a \leq N \leq n$. Finally, we specify the sentence that applies for any natural in A2 to n . Again this should be a little more natural as the free variable n can be confused with the variable bounded to an $\forall$ in A2.

The screenshot shows a Lean proof script for the Sandwich Theorem. The script is as follows:

```

Let n
|b_n - l| < ε

A6 n ≥ N

a_n ≤ b_n by H3 with n
|b_n - l| < ε

A7 a_n ≤ b_n

b_n ≤ c_n by H4 with n
|b_n - l| < ε

A8 b_n ≤ c_n

By inequality properties, N_a ≤ n
|b_n - l| < ε

A9 N_a ≤ n

Let's choose n in A2
|b_n - l| < ε

A10 Lean n ≥ N_a → |a n - l| < ε

```

The script uses the `Let` tactic to introduce a variable n . It then uses `by H3 with n` and `by H4 with n` to derive intermediate goals. The final goal is `Lean n ≥ N_a → |a n - l| < ε`.

Figure 3.26: Sandwich Theorem - 3

Figure 3.27 starts by expressing that the difference between a_n and l is strictly less than ϵ which is obtained by modus ponens with A10 and A9. Then we do very similar steps as previously for sequence c . At the end of this figure, we apply a transformation for inequalities that contain an absolute value property. The transformation is the following $|x - y| < z \Leftrightarrow (x - y < z \wedge y - x < z)$.

The figure shows a Lean theorem prover interface with the following components:

- Hypotheses (Left):**
 - A10: $|a_n - l| < \epsilon$ (by A10 with A9)
 - A11: $|a_n - l| < \epsilon$
 - A12: $N_c \leq n$
 - A13: $n \geq N_c \rightarrow |c_n - l| < \epsilon$
 - A14: $|c_n - l| < \epsilon$
 - A15: $|a_n - l| < \epsilon \wedge |l - a_n| < \epsilon$
- Goal (Right):** $|b_n - l| < \epsilon$
- Proof Steps:**
 - By inequality properties, $N_c \leq n$
 - Let's choose n in A3
 - Let's use absolute value inequality property on A11 A14 and on goal
 - Lean $b_n - l < \epsilon \wedge l - b_n < \epsilon$

Figure 3.27: Sandwich Theorem - 4

Figure 3.28 shows the end of the proof. We first separate the sentences of type $A \wedge B$ in two sentences A and B . We finish by splitting the goal and applying an advanced tactic *By linear arithmetic*⁷. Splitting the goal means at first creating two subgoals that together prove the goal. In here the goal of type $A \wedge B$ is split but as both goals can be proven by linear arithmetic it saves us one line to say to apply the same steps to both subgoals.

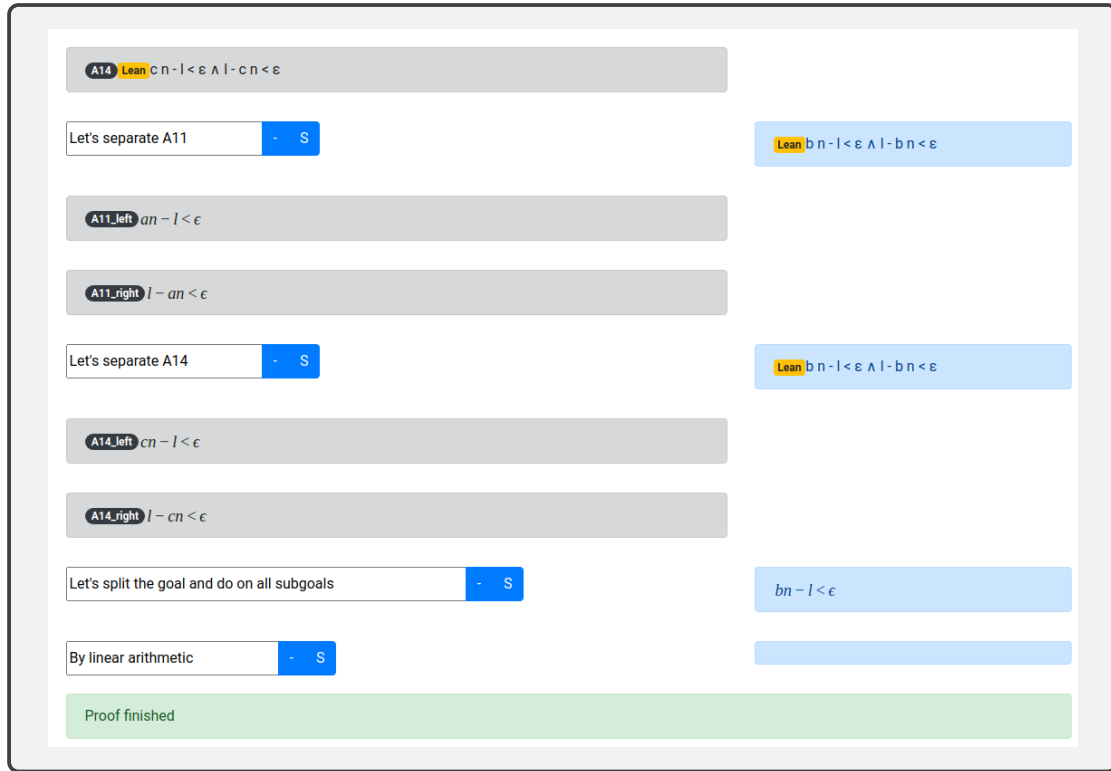


Figure 3.28: Sandwich Theorem - 5

⁷Documentation for that tactic is available at https://leanprover-community.github.io/mathlib_docs/tactic/linarith/frontend.html.

Chapter 4

Future steps and conclusion

This chapter describes some future steps that I could envision for the project. Some of them were considered and explored during the year. A related thesis has been proposed as a subject for academic year 2021-2022. This chapter should give some useful ideas if someone chooses the subject. Should a fellow student retake this work, I would advise to only explore a very limited selection of ideas in this chapter as some require extensive work.

4.1 Translation

This section is about the translation aspect of the system to go from Lean to natural language and back. I believe this is the most difficult part of the project and one key aspect that could benefit from major improvement. I would argue the task is in part hard because writing mathematical proofs stands between real natural language (as English) and a true formal language (as a programming language). Indeed, a rigorous proof is usually much more structured than English but still much less than a programming language.

4.1.1 Machine Learning

One field that could be of great use is machine learning. In particular, one set of tools that could be exploited is the same than used in chatbots. The addition of a python library such as ChatterBot¹ would be interesting. Indeed, the idea would be to classify a given proof instruction to the set of possible instructions and then to output the correct translation in Lean for example. There are however difficulties that oppose the use of Machine Learning. In order to train a classifier, data would be needed. I am not sure what would be a good idea for providing

¹Homepage at <https://github.com/gunthercox/ChatterBot>

such data. Indeed, there is no repository with both mathematical lines of proofs in natural language and their corresponding instructions in Lean. It would then appear that to develop this data one would have to annotate proofs by hand which seems very long to have a good set.

Another aspect that seems difficult to me, is the degree of precision that is required by a translation to be useful. Indeed, for a translation between two languages (for example English and French), sometimes the translation can be a bit erroneous and not sound exactly like a native speaker but still be useful. In this case however, the slightest misinterpretation might lead to severe issues. Still, if it works the inclusion of machine learning would take the translation to a whole new level. Many papers show how machine learning greatly improves translations. It is foreseen by many as the future of natural language translation as discussed in [8].

4.1.2 Community system

Another possibility that could be explored to improve translation would be to create a community driven system to generate new rules. This way instead of having the core developers to write rules (for now mainly as regular expressions) this task would be partly delegated to a community. One could think of an interface that would allow members to write new rules. It would consist of defining an instruction, explaining how it works in natural language and how it should be translated in Lean. As regular expressions are quite difficult to manipulate and to open this rule creation to as many as possible, the system could include a project as [6]. This explores the possibility to write queries of regular expression in natural language. There could then be a way to import rules or to vote for rules of other people as to include them by default. Furthermore, the current system requires priority between the different translation rules as it might happen that two rules work for the same line. This priority could be adapted by the user.

In the same context as described in section 3.1 this feature could be used in a educational context. That is, a teacher could allow only a given set of translation rules to be valid for a specific statement. The teacher could also adapt the priority between the rules.

4.1.3 Lean insights

The system could also be improved on how it gains insights from Lean. Indeed the current implementation does not retrieve all useful information from Lean. Currently, with the VSCode Lean plugin one can peek information on many things by simply selecting them. This would be very useful in a long proof for example to check properties or definition of a given object.

Another aspect that was discussed when presenting Lean is that some advanced features allow to detect possible leads for a proof such as *library_search*. This type of integration is not present in the current implementation. I am not sure how much these advanced features are used in practice when constructing a new proof. Nonetheless, including these strategies gives more power in helping building an unknown proof.

4.2 Mathlib integration

The choice of Lean was mainly driven by the community library *mathlib*. Its integration in the current system might be much more advanced. In particular, the user should be able to import objects from that library in a structured way. For the few theorems proven, the used concepts are already imported. The library contains lots of tactics. One should be able to use them in natural language and thus they would require to be integrated.

4.3 Structured proofs

As mentioned in section 2.3, the ideal system should allow nested proofs. That is, to make a statement and then do the proof of this statement. Lean allows this through the use of the tactic *have* combined with *begin-end* as seen in the code in Figure 2.1. In that example, we prove that the defined element p is a prime number in a subproof. To include these structured proofs in the current implementation, it would be first required to add the frontend functionalities to add proof lines inside other proof lines as shown in Figure 4.1.

Figure 4.1: Example of visual for structured proofs

Then it would be necessary to translate these sub proofs by including *begin-end* instructions.

In the same line of thinking, goals could also be adapted to be written in multiple lines. Indeed, many theorems state a list of consequences. It is of course possible to write a list by writing a goal with the logical and, $\wedge$. However, it would be easier to interact and visualize for the user in a list. One difficulty is that the current implementation does not worry much about the presence of multiple goals. Goals that are expressed as list often need splitting the goal. The implementation would mainly require to make clear to the user that each subgoal is treated at a time.

4.4 Community

Another key aspect that was partially implemented is the community and collaborative aspect of the system. Currently, only the educational system discussed in section 3.1 is available. This should be extended to allow users to see other users

proofs. One could take several ways to extend the current system.

Extending the educational case The educational case could be extended by allowing a grading system by the teacher and by allowing annotations of some steps. Teachers could also fill some of the steps while leaving other steps to the students. They could also split a proof in two parts considering a system that would allow subproofs.

Multiple proofs for same theorem An interesting thing to notice about theorems and proofs is that a theorem can have several proofs. As such, one might create a system in which theorems are made public. When users consider they have a new proof for a given theorem, they might add it to the public list of possible proofs for that theorem. It could then be interesting to highlight what each proof needs as hypotheses or from which set of theories it comes. For example, Pythagorean theorem can be achieved by several ways. One can prove it by geometrical rearrangements but it can also be proven by using calculus. This would increase the educational aspect of the system by encouraging discussions on different views of the same theorem.

Set of theories Another important aspect when building a mathematical knowledge system that is not currently addressed is classification. In order for proofs to be easy to lookup, they should be classified by set of theories. Ideally, one should be even able to see links between theorems. For example, which theorems and previous results are needed to construct a specific proof of the sandwich theorem. This is well handled by the Lean community as seen in the library overview page :

<https://leanprover-community.github.io/mathlib-overview.html>

Interface discoverability One last feature that could be explored is the current interface discoverability. That is, the degree to which the user can easily find how to use the software and in particular how to build a proof. For now the user has to refer to the manual available at :

https://github.com/azarzadavila/mapros/blob/main/user_guide.md

This could be improved by including more possibilities to express the same idea as well as providing suggestions. Each proof line could contain for example a drop-down list of the possible tactics. Another idea could be to use completion by integrating a module similar than these two projects :

- <https://github.com/webscopeio/react-textarea-autocomplete/>
- <https://github.com/moroshko/react-autosuggest/>

4.5 Conclusion

Through this thesis, several aspects of a mathematical proof system were discussed. The current implementation of such a system was presented. It is still far from an ideal system that would allow a large set of inputs from natural language thus being as easy as writing a proof for a mathematical paper. However, it does show a start on how such system could be possible and extremely useful and how it would gain in power by becoming fully collaborative. I do hope that in the coming years systems needing less programming language will arise allowing proofs to be both formal and readable. This chapter describes various improvements to the existing system, hopefully approaching an ideal system.

Appendix A

Screens from the implementation

This appendix contains screens from the implementation that were not included in other chapters as well as a complete proof.



Figure A.1: Menu when not logged as a user

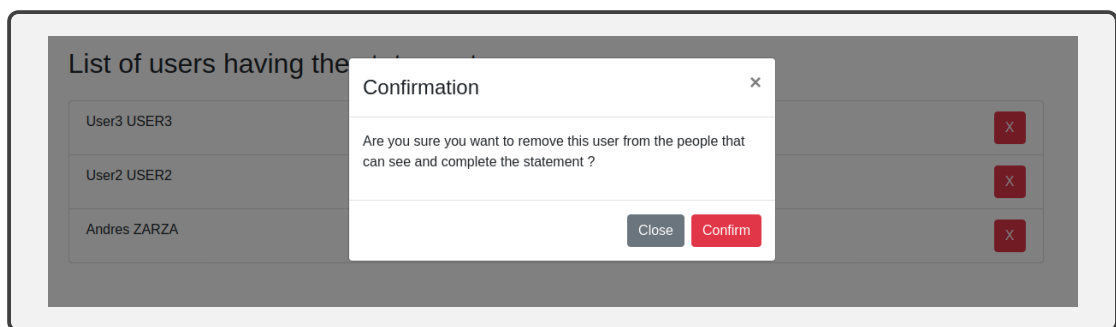


Figure A.2: Confirmation of deletion of user from list of people having a statement

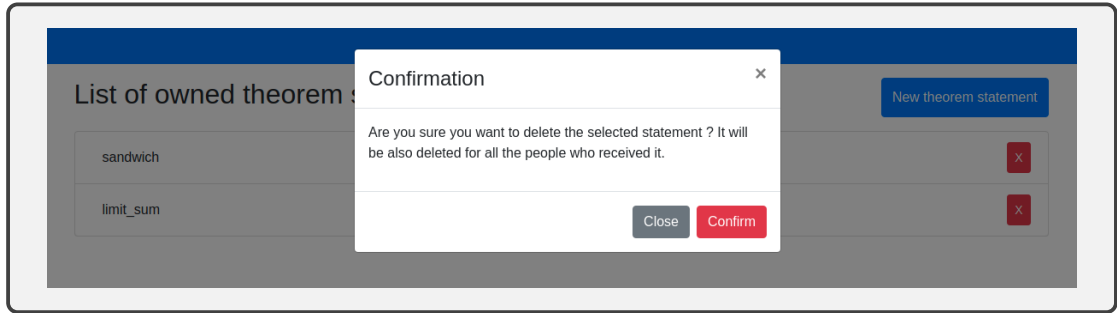


Figure A.3: Confirmation of deletion of statement

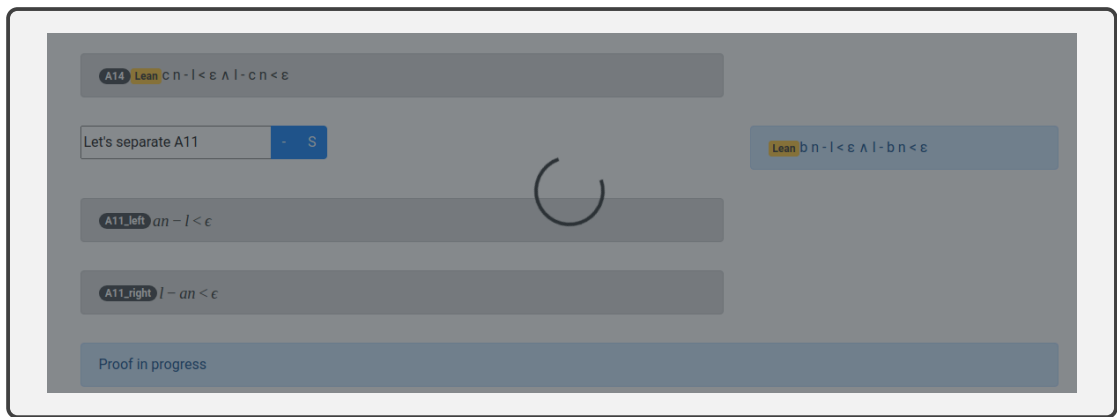


Figure A.4: Animation while awaiting server response

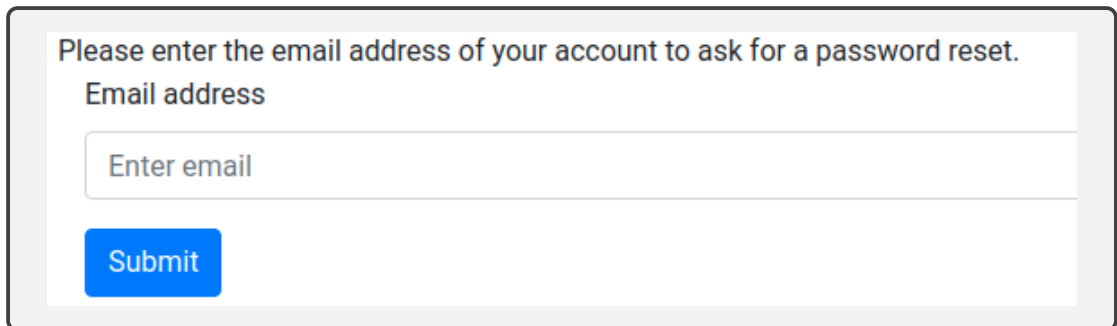


Figure A.5: Ask to reset password

A.1 Limit of a sum example

BACK
Save

Theorem Name :

If,

a_n, b_n are real-valued sequences

$l_a \in \mathbb{R}$

$l_b \in \mathbb{R}$

H1 $a_n \rightarrow l_a$

H2 $b_n \rightarrow l_b$

Then,

$(a + b)_n \rightarrow (l_a + l_b)$

S

Lean `is_limit (a + b) (l_a + l_b)`

Proof:

Let

ε

S

Lean `∃ (N : ℕ), ∀ (n : ℕ), n ≥ N → |(a + b) n - (l_a + l_b)| < ε`

A1 $\epsilon > 0$

Figure A.6: Limit of a sum - 1

By inequality properties,	$\frac{\epsilon}{3} > 0$		- S	$\text{Lean } \exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow (a + b) n - (L_a + L_b) < \epsilon$
A2 $\text{Lean } \epsilon / 3 > 0$				
Let's choose	N_a	such that H1 uses	$\frac{\epsilon}{3}$ (with A2)	$\text{Lean } \exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow (a + b) n - (L_a + L_b) < \epsilon$
A3 $\forall n \in \mathbb{N}: n \geq N_a \Rightarrow an - L_a < \epsilon/3$				
Let's choose	N_b	such that H2 uses	$\frac{\epsilon}{3}$ (with A2)	$\text{Lean } \exists (N : \mathbb{N}), \forall (n : \mathbb{N}), n \geq N \rightarrow (a + b) n - (L_a + L_b) < \epsilon$
A4 $\forall n \in \mathbb{N}: n \geq N_b \Rightarrow bn - L_b < \epsilon/3$				
We claim	$N_a + N_b$	works	- S	$\text{Lean } \forall (n : \mathbb{N}), n \geq N_a + N_b \rightarrow (a + b) n - (L_a + L_b) < \epsilon$
Let	n	- S		
A5 $\text{Lean } n \geq N_a + N_b$				
By inequality properties,	$n \geq N_a$	- S	$\text{Lean } (a + b) n - (L_a + L_b) < \epsilon$	

Figure A.7: Limit of a sum - 2

A6 $n \geq N_a$		
By inequality properties,	$n \geq N_b$	- S
		Lean $ (a+b)n - (L_a + L_b) < \varepsilon$
A7 $n \geq N_b$		
Let's choose n in A3	- S	Lean $ (a+b)n - (L_a + L_b) < \varepsilon$
A8 Lean $n \geq N_a \rightarrow a n - L_a < \varepsilon / 3$		
By A8 with A6	- S	Lean $ (a+b)n - (L_a + L_b) < \varepsilon$
A9 Lean $ a n - L_a < \varepsilon / 3$		
Let's choose n in A4	- S	Lean $ (a+b)n - (L_a + L_b) < \varepsilon$
A10 Lean $n \geq N_b \rightarrow b n - L_b < \varepsilon / 3$		
By A10 with A7	- S	Lean $ (a+b)n - (L_a + L_b) < \varepsilon$
A11 Lean $ b n - L_b < \varepsilon / 3$		

Figure A.8: Limit of a sum - 3

Let's use absolute value inequality property on A9 A11 and on goal

S

Lean $(a + b) n - (l_a + l_b) < \varepsilon \wedge l_a + l_b - (a + b) n < \varepsilon$

A9 Lean $a n - l_a < \varepsilon / 3 \wedge l_a - a n < \varepsilon / 3$

A11 Lean $b n - l_b < \varepsilon / 3 \wedge l_b - b n < \varepsilon / 3$

$(a + b)_n = a_n + b_n$

by definition of addition for functions

S

Lean $(a + b) n - (l_a + l_b) < \varepsilon \wedge l_a + l_b - (a + b) n < \varepsilon$

A12 Lean $(a + b) n = a n + b n$

Let's split the goal and do on all subgoals

S

Lean $(a + b) n - (l_a + l_b) < \varepsilon$

By inequality properties

S

Proof finished

+

Figure A.9: Limit of a sum - 3

Bibliography

- [1] K. Appel and W. Haken. Every planar map is four colorable. part i: Discharging. *Illinois J. Math.*, 21(3):429–490, 09 1977.
- [2] Robert S. Boyer. A mechanically proof-checked encyclopedia of mathematics: Should we build one? can we? In Alan Bundy, editor, *Automated Deduction — CADE-12*, pages 237–251, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
- [3] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.
- [4] Thibault Gauthier and Cezary Kaliszyk. Aligning concepts across proof assistant libraries. *Journal of Symbolic Computation*, 90:89–123, 2019. Symbolic Computation in Software Science.
- [5] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, Mar 2012.
- [6] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. Neural generation of regular expressions from natural language with minimal domain knowledge. *CoRR*, abs/1608.03000, 2016.
- [7] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.
- [8] Zhixing Tan, Shuo Wang, Zonghan Yang, Gang Chen, Xuancheng Huang, Maosong Sun, and Yang Liu. Neural machine translation: A review of methods, resources, and tools. *AI Open*, 1:5–21, 2020.
- [9] Floris van Doorn, Gabriel Ebner, and Robert Y. Lewis. Maintaining a library of formal mathematics. *Intelligent Computer Mathematics*, page 251–267, 2020.

- [10] Alan Weir. Formalism in the Philosophy of Mathematics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2021 edition, 2021.
- [11] Ittay Weiss. The qed manifesto — version 2.0. In *Asia-Pacific World Congress on Computer Science and Engineering*, pages 1–7, 2014.
- [12] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica; 2nd ed.* Cambridge Univ. Press, Cambridge, 1927.
- [13] Bohua Zhan. holpy: Interactive theorem proving in python. *CoRR*, abs/1905.05970, 2019.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl