UCLouvain

École polytechnique de Louvain

A parser for auto-formalization in education

Author: **Augustin D'Oultremont**
Supervisor: **François Glineur**
Readers: **Peter Van Roy, Sébastien Mattenet**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Abstract

This project is a proof of concept for the use of a parser for auto-formalization in education. The aim was to build an easy-to-expand translation system for mathematical proofs, translating natural language proofs to machine-verifiable `lean4` code, statement by statement.

One example of such a statement is "*Considering the different possibilities of the modulo operation, we have $q \mod 3 \in \{ 0, 1, 2 \}$.*". This translation works by identifying keywords such as "*have*" to extract the meaning of the sentence, while other "filler" words are ignored.

This allows us to write proofs in a way that is readable and feels natural, although knowledge of the keywords is very important. We managed to create a system that will allow us to easily add support for real numbers, inequalities, and justifications when they will be supported in `lean4`.

Acknowledgements

I would like to thank Sébastien Mattenet for his time and availability during the elaboration of this thesis, as well as Prof. François Glineur for his feedback.

A special thank-you goes to my friends for supporting me throughout this project, especially Alix de Pierpont, Brieuc de Voghel, and Gwenaëlle Dembour, who have supported me and helped me get back on track when feeling unmotivated.

I would also like to thank my parents for supporting me throughout my five years of university, and well before that. A special thanks goes to my dad, Gilles d'Oultremont, for taking time to read my thesis during the busy period of a move.

A very special thank-you has to go to Valentin Lemaire, whose input during the whole project, from coding to writing, as well as his friendship and assistance could not be matched.

Contents

CHAPTER 1

Introduction

1.1 The problem with teaching proofs and proving

Most theorems can be proven in many ways, but ensuring that a proof is correct can be quite difficult. Currently, proof verification relies either on external review to detect errors in the reasoning, or on proof assistants, which verify proofs automatically. These two methods both have their strengths and weaknesses.

External review is done by another mathematician, a teacher or a different student checking for errors in the reasoning, while proof assistants use rules of inference to automatically and mechanically check the validity of statements.

On the one hand, external review has the advantage of allowing one to write proofs the "usual" way, using natural language, but at the expense of taking time and relying on the reviewer to detect the errors, which does not guarantee correctness. On the other hand, proof assistants can guarantee proof correctness, but use strict syntax, and are usually harder to learn.

In the case of university students learning to prove theorems, the large number of students with relatively limited prior knowledge make the flaws of both systems critical. External review cannot be done by other students, as they are not used to writing proofs and would have a hard time detecting errors or lack of rigor. If it is done by teachers or teaching assistants, it quickly becomes a logistical problem; the large number of students and large number of exercises needed to learn make this practically impossible.

The last possibility, using proof assistants, is usually too complex for introductory courses, especially when the aim is to teach theorem proving in general.

This results in most students learning to prove theorems by trying to get to the teacher's solution. This contrasts with the fact that there are many ways to prove a statement, and students sometimes overlook small details that could be necessary to be rigorous. It is therefore important to find a solution to give feedback to students automatically.

1.2 How to automate feedback ?

The idea is to create a translation system between natural language and formal proofs (using the strict syntax of a proof assistant). This would allow students to write exercise proofs in natural language, translate it automatically and obtain feedback that will tell them whether the proof is correct. The general problem of translating natural language proofs into a formal language is called *auto-formalization*, and is a relatively recent[1] subject of research.

1.3 Our solution

For our use case, we need to take into account that a significant part of the input sentences will be wrong and that the system should be explainable, as students need to know how they can formulate their statements without too much trial-and-error. These requirements rule out machine learning techniques used in most current research, as they are not deterministic and lacks explainability, and drives us towards the use of parsing. Although not an easy solution either, parsing allows us to better control the translations and accepted input sentences.

This master's thesis is based on Andres Zarza Davila's work [2] conducted in 2020-2021. He analyzed different proof assistants and choose `lean3` as his target language. He created a web interface around his translation system (including database and user login) but unfortunately didn't have the time to finish these features completely.

With the recent release of `lean4`, we decided to switch to this version, which unfortunately is still in development and missing many important features. The working examples are therefore very simple and closer to high-school level than to our target audience (university students). The use of `lean4`, the structure of Andres' work, and

[1] The earliest paper we found is a PhD Thesis from 2012 [1].

the fact that it had many unnecessary features (more information in sections 2.4.1 and 2.5) made us start from scratch and try to build a more modular system.

Our focus was therefore to create a translation system that could be easily be used as a building block for a larger system. This document explains how the program works, along with its limitations and what improvements could be made.

To showcase this and make it usable, we also created a very basic command line interface (CLI). This project is therefore split in 3 GitHub repositories: the main program (`natural2lean`)[2], the command line interface (`natural2lean-cli`)[3] and the project template[4] (used to get access to a `lean` project containing some utilities).

[2] `https://github.com/Augustindou/natural2lean`
[3] `https://github.com/Augustindou/natural2lean-cli`
[4] `https://github.com/Augustindou/natural2lean-lean-project-template`

CHAPTER 2

Related Work

In this chapter, we will give a general explanation of formal proofs and the auto-formalization problem and explore a few related works applying machine learning and parsing to the problem.

2.1 Formal proofs and auto-formalization

Formal proofs are structured[1] and machine verifiable[2]. Each statement is derived from previous ones, from assumptions, or from axioms using rules of inference. Systems verifying these formal proofs are called *proof assistants*. Some of the most popular are *Coq* [3], *Isabelle* [4], *Mizar* [5], and *Lean* [6].

These proof assistants have the ability to confirm that a proof is correct, but rely on a strict syntax and sometimes use different ways of structuring proofs. This can make them hard to learn and not especially useful when teaching the reasoning behind theorem proving, while making the written proofs hard to read.

On the other hand, natural language proofs (not written with the specific syntax of a proof assistant) have the disadvantage of not being verifiable while usually being more readable than their formal counterparts. Many mathematicians are also familiar with

[1] Structured proofs are a list of statements where each statement can be derived from previous statements or axioms using rules of inference.

[2] Machine verifiable proofs can be verified automatically by a computer program, called proof assistant.

natural language proofs and write their proofs in that way, while only a proportionally small number of them tackle formal proofs.

To bridge the gap between formal and informal proofs, some researchers want to automate the translation between the two. Translation from formal to informal is a relatively easy task, as the formal proofs are more detailed and have less variation. Some programs already do that fairly well, such as the Mizar-to-LaTeX translator [7, 8]. The following sections describe some attempts at the more difficult task of translating informal proofs to formal ones (a process called *auto-formalization*).

2.2 Proving in lean

Lean[3] relies on hypotheses and tactics to prove theorems[4]. A hypothesis is a proposition that has been assumed or derived from assumptions[5]. It has either been assumed, or was derived from other hypotheses or from axioms using rules of inference. Tactics allow us to manipulate these hypotheses to get to our goal (the proposition we want to prove).

The basic tactics are quite simple; in the example below, we first used the "`rw`" tactic, which rewrites a hypothesis. In this case, it replaces the a in hypothesis h_0 by 2 (because hypothesis h_1 tells us that `a = 2`). The "`apply`" tactic then transforms our goal from `b = 2` to `2 = b` using the symmetry of the equality operator, which lets us close the goal with the "`assumption`" tactic, checking for a hypothesis that is the same as the goal.

```
example (a b : Nat) (h₀ : a = b) (h₁ : a = 2) : b = 2 := by
  rw [h₁] at h₀ -- we get h₀ : 2 = b
  apply Eq.symm -- we now need to prove 2 = b
  assumption -- closes the goal because h₀ is 2 = b
```

This can seem quite complicated and verbose, but more complex tactics can be written, such as "`simp_all`", which rewrites hypotheses and simplifies them to get to a conclusion. This tactic allowed us to automatically solve the goal in the example below.

```
example (a b : Nat) (h₀ : a = b) (h₁ : a = 2) : b = 2 := by
  simp_all
```

[3] This is only a small explanation of how `lean` works, but much more information can be found online

[4] Although it is also possible to prove theorems by creating functions instead of using tactics, this representation of proofs is quite unusual. We will therefore focus our attention on tactics.

[5] This is a broader term than the mathematical hypothesis, which only includes assumptions.

One similar tactic is "`ring`", which uses ring theory to solve equations, and another useful tactic is "have", which introduces a new hypothesis. More complex examples are given in appendix B, where we show theorems that have been translated automatically by our system.

2.3 Machine Learning

Machine learning being a widely used and versatile tool in neural translation, it is not surprising that researchers found it to be useful in natural to formal math translation. Although very powerful, it comes with a few drawbacks; the need for large amounts of data and the stochastic aspect.

As stated in *First Experiments with Neural Translation of Informal to Formal Mathematics* [9], one of the difficulties with applying machine learning techniques to autoformalization is the very limited amount of aligned informal-formal data. These corpora would allow the models to learn the correspondence between the formal and informal proofs, but the lack of them requires us to find other ways to get data.

Different approaches were developed. *Informalization* (sections 2.3.1 and 2.3.2) was used in the beginning for its capacity of creating large amounts of aligned formal-informal data, but more recently, the using the power of large language models such as OpenAI's Codex [10] seems to be the best method (as shown in sections 2.3.4 and 2.3.5).

2.3.1 First experiments

In *First experiments with neural translation of informal to formal mathematics*, Qingxiang Wang, Cezary Kaliszyk, and Josef Urban [9] circumvent the lack of aligned corpus by using *informalization* (also called *ambiguation*). This is a group of techniques that transform formal sentences into less formal counterparts. These techniques allowed them to create a corpus of informal to formal sentence mappings, which can then be used to train the network.

They used the Mizar system [11] as a source, which is a proof assistant and formal language having one of the biggest libraries of formal mathematics. They also used the already existing Mizar translation to LaTeX [7, 8], which is relatively non-trivial. We can see in table 2.1 that this translation is quite verbose, but similar to what one would write in LaTeX.

Rendered LaTeX	Suppose s_8 is convergent and s_7 is convergent. Then $\lim(s_8+s_7) = \lim s_8 + \lim s_7$
Mizar	`seq1 is convergent & seq2 is convergent implies lim(seq1+seq2)=(lim seq1)+(lim seq2);`
Tokenized Mizar	`seq1 is convergent & seq2 is convergent implies lim (seq1 + seq2 ) = ( lim seq1 ) + ( lim seq2 ) ;`
LaTeX	`Suppose ${s_{8}}$ is convergent and ${s_{7}}$ is convergent. Then $\mathop{\rm lim}({s_{8}}{+}{s_{7}}) \mathrel{=}\mathop{\rm lim}{s_{8}}{+} \mathop{\rm lim}{s_{7}}$`
Tokenized LaTeX	`Suppose $ { s _ { 8 } } $ is convergent and $ { s _ { 7 } } $ is convergent . Then $ \mathop { \rm lim } ( { s _ { 8 } } { + } { s _ { 7 } } ) \mathrel { = } \mathop { \rm lim } { s _ { 8 } } { + } \mathop { \rm lim } { s _ { 7 } } $`

Table 2.1: Example of informalization using the Mizar translation to LaTeX[9]

To evaluate their model, they used the *BiLingual Evaluation Understudy* (BLEU) score, a widely-used metric for evaluating machine-translated text. This article from Google [12] rates anything higher than 60 as *Quality often better than human*, and anything below 10 as *almost useless*.

Using this method of informalization to create the data, the best BLEU score they managed to obtain was 69.5, but we should keep in mind that a reverse translation was used to obtain the data, which might therefore have a structure and a wording that allow the neural network to perform better than it would with real-world data (e.g., a mathematician would probably not write s_8 as `${s_{8}}$`).

2.3.2 Follow-up

The same team then published *Exploration of Neural Machine Translation in Autoformalization of Mathematics in Mizar* [13], diving deeper into the subject. They kept the same approach to generate the data, for which they obtained a slightly improved 70.9 BLEU score, but also tried it on the ProofWiki — Mizar dataset, where they managed to obtain BLEU scores of around 6 to 8. This shows that the method of generating the data from the formal proofs gives better performances, probably due to a mix of the data being closer to the expected output because of informalization, but also to the access to a much larger dataset. We also have to keep in mind that the BLEU score is intended for text translation, and not math translation.

2.3.3 `MiniF2F`

In February 2022, OpenAI published a paper presenting `MiniF2F` [14] a dataset of a few hundred Olympiad-level problem statements formalized for different theorem provers. The aim was to create a benchmark to evaluate neural theorem proving across theorem provers. As explained in section 2.3.4, it can also be used to evaluate auto-formalization systems, creating a much more accurate score than the ones given in the first two papers (sections 2.3.1 and 2.3.2), which used the BLEU metric on data obtained by informalization to evaluate their models.

2.3.4 Large Language Models

In a very recent paper called *Autoformalization with Large Language Models* [15] (published on May 25th, 2022), a team from Google Research and the University of Cambridge used large language models to perform auto-formalization. These models are very versatile and only need a small amount of data to be tuned to their intended task.

To evaluate their model, they used OpenAI's miniF2F [14] and they managed to bump the state-of-the-art proof rate from 29.6% to 35.2% using *few-shot learning*[6] in combination with large language models (in particular Google Research's PaLM [16] and OpenAI's Codex [10]). This cannot be compared to a BLEU score, but this is closer to a real-world application than the previous papers.

2.3.5 Chat interface

Many Lean developers are also hard at work on this subject, such as Zhangir Azerbayev and Edward Ayers, who created an OpenAI Codex-powered chat interface for formalizing statements, discussed on the Lean community forum [17].

2.4 Parsing

Parsing is another possibility to implement a translation system, with the advantage of being predictable, but the disadvantage of needing to predict every input possibility, as sentences can be written in many ways in natural language.

2.4.1 MaProS

This master's thesis is based on Andres Zarza Davila's MaProS [2]. He analyzed different proof assistants and chose `lean3` as the best for his use case. He created a web user interface, along with a database and a user login system, and showcased his system with a proof of the *Sandwich Theorem* as well as a proof about the limit of a sum of two sequences.

2.5 Auto-formalization in education

The need for an explainable and predictable system ruled out machine learning for our use case, leaving us with the aim to create a parser. We quickly realized that expanding on MaProS would be quite difficult. Due to lack of time, many features (database, users, ...) were not quite finished, and the imposed syntax was very rigid and not very expandable, as it matched complete sentences and lacked modularity.

Starting from scratch allowed us to use `lean4`, which is more powerful and solves many shortcomings of `lean3`. Unfortunately, we did not evaluate well at the time how many features were missing from the new `lean4`. We also focused on a core translation

[6] Few-shot learning is a machine learning technique developed with the aim to build accurate machine learning models with a small amount of training data.

system instead of making a full program, to allow it to be used by other programs and allow programmers to build upon that afterwards.

CHAPTER 3

System overview

In this chapter, we will explain how the system works from a user's point of view. The screenshots shown are from the `natural2lean-cli`, but the core translation system can easily be used in another system (using the `natural2lean Translator` detailed in chapter 4).

3.1 Keywords

The system works by matching keywords recursively. This section describes the patterns used to match keywords. The system does not work with a rigid grammar, as the aim was to keep the input as unstructured as possible. Therefore, when the keywords are matched, many parts of the input are ignored and the relative positions of keywords are not always fixed.

3.1.1 Algebraic expressions

Algebraic expressions are one of the main building blocks of the `natural2lean` translation system. Every math expression (encapsulated by "$" symbols in LaTeX) is first translated to the lean format (as shown in example 3.1) and then parsed as an equation, a split into cases, one or more identifiers, or an expression. All of these terms are described below.

LaTeX	`$x ^ { \frac{1}{2} } + 2x$`
Lean	`x ^( ((1) / (2)) ) + 2*x`

Example 3.1: Translation of an algebraic expression from LaTeX to lean

Identifiers

Identifiers are variable names. They need to start with a letter, but can contain any combination of letters, numbers, and underscores. Multiple identifiers can also be grouped in one single expression and their type can be defined in algebra or in text. In case of the latter, the proposition system translates it to the algebraic version. This will be explained in section 3.1.2. Examples of these can be found in example 3.2.

Single identifiers	`$x$, $x1$, $x_1$`
Multiple identifiers	`$x, y$, $x_1, x_2, x_3$`
Identifiers in a set	`$x, y \in \mathbb{N}$`
Text-defined set	`$x, y$ are` **`natural`** `numbers`

Example 3.2: Definition of different identifiers / variables

Expressions

Expressions can essentially match any algebraic expression, but due to the matching order, they will only match numbers or sequences of math operations without comparison operators (e.g., anything containing =, <, ... will be matched as an equation before getting the chance to be matched as an expression). Some examples are shown in example 3.3.

```
$x+2$
$x ^ { \frac{1}{2} } + 2x$
```

Example 3.3: Expressions

Equations

Equations are combinations of expressions linked by relational operators. Currently, the only supported operator is the equality sign, as there are no powerful lean4 tactics yet that deal with inequalities (this is described in section 5.1). Equations are translated by taking the left and rightmost expressions, and expressions in the middle are given as indications to lean on which important steps to take in the expansion. This expansion can use already proven equalities, as shown in example 3.4.

Already proven	$a = 2k$
Input	`$a^2 = (2k)^2 = 4k^2$`
Final translation	$a^2 = 4k^2$
Steps	$a^2 = (2k)^2$
	$(2k)^2 = 4k^2$

Example 3.4: Translation of an equation

Cases

Cases are the last algebraic expression currently implemented. Stating that an expression belongs to a finite set is translated as a disjunction on the values of x and directly split into different cases (see example 3.5). Finite sets expressed as $\{1, 2, \ldots, 10\}$ or $\{1, 2, \ldots, n\}$ are not supported. All the elements of the set have to be expressed explicitly (e.g., $\{1, 2, 3, 4, 5\}$).

Input	`$x \mod 3 \in \{0, 1, 2\}$`
Translation	$x \mod 3 = 0 \vee x \mod 3 = 1 \vee x \mod 3 = 2$
Produced cases	prove theorem assuming $x \mod 3 = 0$
	prove theorem assuming $x \mod 3 = 1$
	prove theorem assuming $x \mod 3 = 2$

Example 3.5: Translation of possibilities for a variable

3.1.2 Propositions

A proposition is a building block that can be used in many statements. They can be simple such as "x is even" or "y is divisible by 3", but can also be negated or combined, forming implications or conjunctions, which are also propositions. Propositions also include equations and separations into cases mentioned in section 3.1.1[1].

Combinations of propositions

These propositions can be combined with "and" or a simple comma, but it is usually recommended to use smaller steps and divide these in multiple lines. Combining propositions is mostly useful in a theorem statement, where the different hypotheses can be used separately (see section 3.1.3 and example 3.6).

[1]Although this is not the case in the code, an end user can consider equations and separations into cases as propositions.

Input	`$a$ and $b$ are even and $c$ is odd`
Separation	a is even
	b is even
	c is odd
Translation	a is even $\wedge$ b is even $\wedge$ c is odd

Example 3.6: Parsing of a sentence containing multiple propositions

Implications

Implications use the "if" and "then" keywords to separate the hypotheses from the theses. These were mainly used in theorem statements (more information can be found in section 3.1.3). A simple example of an implication is given below (example 3.7), but note that without the "`\cdot`", the system will assume that `xy` is a single variable.

Input	`if $x$ and $y$ are even, then $x \cdot y$ is even`
Hypotheses	x is even
	y is even
Thesis	$x \cdot y$ is even

Example 3.7: Decomposition of an implication

Functions

Functions are used to describe certain properties of expressions. For now, the only existing functions are "even", "odd" and "divisible by", but more can be added easily. These functions can also be negated, if "not" or "n't" is present in the proposition.

The patterns that can be used for "even" and "odd" is "`$ [arg] $ [...] [even/odd]`"[2]. Therefore, the expressions in example 3.8 can both be used to tell the system that the variable x is even. Similarly, the pattern for "divisible" has two arguments: "`$ [arg] $ [...] divisible by [...] $ [arg] $`".

`$x$ is even`
`$x$ is an even number`

Example 3.8: Equivalent statements for evenness of a variable

[2] "`[...]`" can contain other words that are ignored by the system.

The modularity baked into the system allows us to apply one or more functions to one or more variables (see example 3.9). It is also in this part of the system that sets defined in text are translated to their algebraic counterparts (see example 3.10).

Input	**a and b** are **even** numbers **divisible by 3 and c** is **not divisible by 5**
Separation	a is even b is even a is divisible by 3 b is divisible by 3 c is not divisible by 5

Example 3.9: Multiple functions applied to multiple variables

Input	**a and b** are **even natural** numbers
Separation	$a, b \in \mathbb{N}$ a is even b is even

Example 3.10: Combining functions and text-defined sets

Such that

A "such that" statement conveys the meaning of "$\exists x, P(x)$" in mathematics. The pattern that it recognizes is "[left part] such that [right part]" and will then try to find one or more identifiers in the left part, as well as any proposition in the right part. Example 3.11 shows a typical example of how such that can be used.

Input	By **definition** of the modulo operation, we **have $k \in \mathbb{N}$ such that $q = 3k + 1$**.
Output	$\exists k \in \mathbb{N}, q = 3k + 1$

Example 3.11: Such that

3.1.3 Theorem statements

The basic building blocks explained in section 3.1.1 and 3.1.2 can be used to state the theorems that we want to prove. This can be done in the two ways described below.

Named theorem

A named theorem statement is a sentence of the form "`[theorem/lemma] [name] : [statement]`". The statement should either be an implication or one or more propositions (see example 3.12).

```
theorem even number squared : if $ m \in \mathbb{N}$ is even, then
$m^2$ is even.
```

Example 3.12: Named theorem statement

Unnamed theorem

For quick proof testing, one can skip the keywords and name and simply give the theorem statement, creating an unnamed theorem[3] (shown in example 3.13). The main advantage is that it allows us to write less text, but the user will not be able to use the proved theorem in another proof.

```
for any natural number $n$, $2n+4$ is even.
```

Example 3.13: Unnamed theorem

Using a theorem

If a theorem has been named, it can be used in a future proof by simply mentioning its name. This allows for a lot of flexibility, giving the user the possibility to prove simple lemmas separately.

3.1.4 Statements

Each proof consists of a list of statements, which are the sentences given as input. These statements are recognized with specific keywords, but some filler words can be added to make the sentences readable.

Have

The "have" statement is the most important and most versatile statement in the system. The aim was to recognize sentences that indicate a step of reasoning (example

[3]This can be referred to as "Example" in error messages or in the code.

3.14 shows some of them), but since this type of sentence can vary a lot, trying to account for all the possibilities is not easy. Therefore, the structure of the sentence is "`[left part] [keyword] [right part]`" and the keyword can be any one of "have", "derive", "gives", "show" and "prove".

```
By expansion, we have $n^2 = (2k)^2 = 4k^2$
From that, we can easily derive $n^2 = (2k)^2 = 4k^2$
This gives us $n^2 = (2k)^2 = 4k^2$
By expansion, we show that $n^2 = (2k)^2 = 4k^2$
We can easily prove that $n^2 = (2k)^2 = 4k^2$
```

Example 3.14: Equivalent formulations giving the same result

The right part can be any proposition or a simple statement. This is what the system will try to verify, and, as stated before, the propositions include equations and separations into cases.

If an expansion is not too complex and only relies on previous statements, it should work without indicating anything, but other statements could need any of the following keywords (these can be present in any part of the sentence). Example 3.15 shows some use cases.

- previously proven theorems can simply be mentioned in the sentence,
- **induction hypothesis** if a statement relies on a base case of the current induction,
- **`definition`** if it can easily be deduced from another statement,
- **`possibilities [...] modulo` or `modulo [...] possibilities`** only works for values below 5[4], but can be used to justify `$n \mod 3 \in \{0, 1, 2\}$`. Another limitation is that the different values have to be in order ($\{0, 1, 2\}$ and not $\{0, 2, 1\}$).

[4] Due to current `Mathlib4` limitations, they had to be manually added. We limited this to 5 because splitting a case into more than 5 possibilities seemed impractical. More information can be found in section 5.1.

Using other proof	`By the "square mod 3" theorem, we have $q^2 \mod 3 = 1$`
Induction hypothesis	`The induction hypothesis gives us that $k^3 + 2k$ is divisible by $3$.`
Definition	`By definition of the modulo operation, we have $k \in \mathbb{N}$ such that $q = 3k + 1$.`
Modulo	`Considering the different possibilities of the modulo operation, we have $q \mod 3 \in \{ 0, 1, 2 \}$.`
Keyword at the end	`We have that $k^3 + 2k$ is divisible by $3$, which is our induction hypothesis.`

Example 3.15: Different helper words for "have" statements

Induction

An induction is detected by the pattern "`[...] induction [...] $ [arg] $ [...]`" where the argument is the variable on which to do the induction. One can then specify which case we want to develop by stating "`$ [arg] = [value] $`", and a sub-statement can then be added (another induction, a have statement, defining a proof structure, or a simple statement). The induction hypothesis has to be specified if it is needed.

Example 3.16 shows a complete working proof using induction. Note that for the conclusion, only one part is needed, either telling the system that $(k+1)^3 + 2(k+1)$ is divisible by 3, or telling the system that $n^3 + 2n$ is divisible by 3 will finish the proof.

Contradiction & Contraposition

These two proof structures are simple indications of how we will proceed. To use them, simply mention the word in the input. Due to the matching priority, if your sentence contains other keywords, it might be mismatched.

Example 3.17 shows the beginning of a proof by contradiction. We can see that after stating that we will use contradiction, we have assumed that $2n$ is not even, and we need to prove "False", which is lean's way of telling us that we need to show a contradiction.

Similarly, example 3.18 shows how we can use the `contraposition` keyword. After stating that we will prove the contrapositive, The system asks us to prove that n is not

Theorem statement	For any **natural** number **n**, **$n^3 + 2n$** is **divisible by 3**.
Induction	We will use **induction** on **n**.
Base case	The base case, **$n = 0$**, is **trivial**.
Start induction step	For **$n = k+1$**, the **induction hypothesis gives** us that **$k^3 + 2k$** is **divisible by 3**.
Development	Therefore, we **have** a **natural** number **z such that $k^3 + 2k = 3z$**.
Development	By expansion, we can **derive $(k+1)^3 + 2(k+1) = k^3 + 3k^2 + 5k + 3 = k^3 + 2k + 3(k^2 + k + 1) = 3z + 3(k^2 + k + 1) = 3(z + k^2 + k + 1)$**.
Conclusion	Hence, **$(k+1)^3 + 2(k+1)$** is **divisible by 3**, therefore $n^3 + 2n$ is divisible by 3.

Example 3.16: Complete proof using induction

Theorem statement	For any **natural** number **n**, **$2n$** is **even**.
Feedback given	To finish this proof, we need to prove this: even (2 * n) Variables we can use : n : $\mathbb{N}$
Statement	We will use **contradiction**.
New feedback	To finish this proof, we need to prove this: **False** Variables we can use : n : $\mathbb{N}$ What we have shown so far (for this goal) : **h_0 : ¬even (2 * n)**

Example 3.17: Proving by contradiction

even, based on the fact that n^2 is not even instead of proving that n^2 is even if n is even. The `contraposition` works by inverting the last assumption or proved statement, so creating a new statement with "have" right before doing the `contraposition` allows one to choose what to prove.

Theorem statement	**If n** is an **even natural** number, **then n^2** is also **even.**
Feedback given	To finish this proof, we need to prove this: even (n ^ 2) Variables we can use : n : $\mathbb{N}$ What we have shown so far (for this goal) : h_0 : even n
Statement	We will prove the **contrapositive**
New feedback	To finish this proof, we need to prove this: ¬**even n** Variables we can use : n : $\mathbb{N}$ What we have shown so far (for this goal) : h_0 : ¬**even (n ^ 2)**

Example 3.18: Proving by contraposition

Simple statements

Simple statements can be used to solve small cases, such as a base case in an induction (see the base case of example 3.16) or for a simple case after a split into different possibilities (see example 3.19). Currently, `contradiction` and `trivial` are supported. `contradiction` works if two things that the user has shown directly contradict themselves (as is the case in example 3.19, where h_0 and h_1 contradict each other) and `trivial` works best when the result is obvious (in example 3.16, it is used to prove that $0^3 + 2 \cdot 0$ is divisible by 3).

3.1.5 Combining these elements

All these building blocks can be combined to create sentences. One example of this is shown in figure 3.1. When writing a statement, the system checks for these, in order:

1. a specific case when proving a theorem by induction,
2. the start of an induction

Theorem statement	**If q** is a **natural not divisible by 3**, **then $q^2 \mod 3 = 1$**.
Splitting	Considering the different **possibilities** of the **modulo** operation, we **have $q \mod 3 \in \{ 0, 1, 2 \}$**.
Solving first case	For the first case, as q is not divisible by 3, we **have** a **contradiction**.
What it solved	1st case: solve q ^ 2 % 3 = 1 Variables we can use : q : ℕ What we have shown so far (for this goal) : h_0 : ¬divisible 3 q h_1 : q % 3 = 0

Example 3.19: Solving a goal by contradiction

3. a "have" statement
4. a contradiction / contraposition proof structure,
5. a proposition or a simple statement, only to conclude the proof.

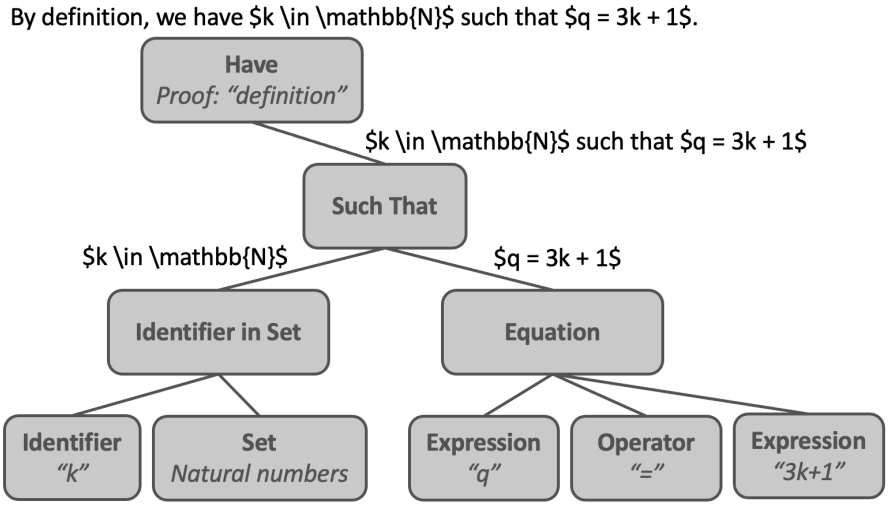

Figure 3.1: Combination of elements to form a sentence

3.2 Interaction with the system

3.2.1 Installation

Installing the project can be done by following the few steps below[5]. Note that the program might take about 10 minutes to start on first launch, as it has to download `lean4` and `mathlib4`, which are both large projects.

1. Make sure **`python3`** (version `3.9` or higher) is installed on your system[6].
2. Install **`elan`**, the lean version manager; the latest version can be found at `https://github.com/leanprover/elan/releases`.
3. Install the **`natural2lean`** command line interface by running "`pip install natural2lean-cli`".
4. On Windows, you might need to add something to the path for the `natural2lean` command to work. Look out for warnings when running "`pip install`".

3.2.2 Launching the CLI

The command line interface supports multiple ways to launch it. After installation, using the `natural2lean` command will launch the program, asking the user for the mode (as shown in figure 3.2). One can then choose between `interactive` mode (explained in section 3.2.6) or `file` mode (explained in section 3.2.5).

Figure 3.2: Choosing the mode in the CLI

If the file mode is chosen, the system then asks the user for a file name or path (showed in figure 3.3), before checking the file line by line as explained in section 3.2.5.

3.2.3 Shortcuts

For ease of use, command line arguments have been added.

[5]This part focuses on `natural2lean-cli`, but `natural2lean` has the same requirements.
[6]More information can be found at `https://www.python.org/downloads/`.

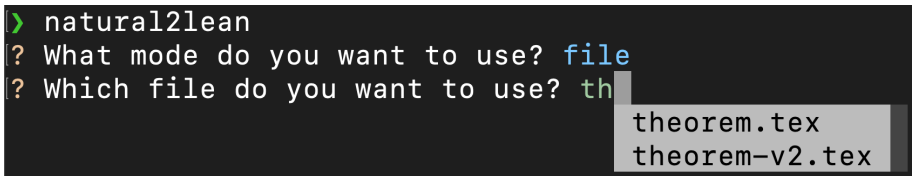

Figure 3.3: Giving the file name to the system

File mode

Using "`file`" or "`f`" will skip the first prompt and ask directly for the input file, which can also be specified as an argument (examples shown in figures 3.4 and 3.5).

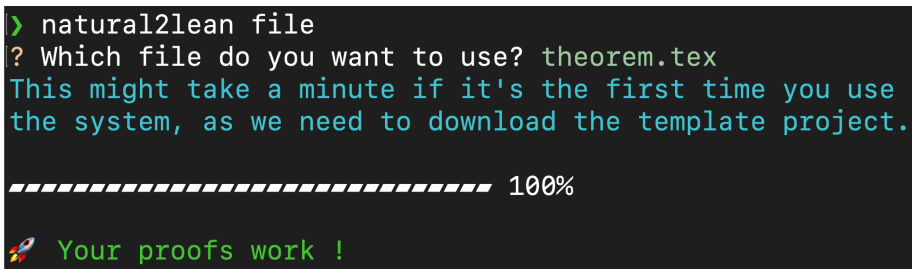

Figure 3.4: Shortcut for file mode

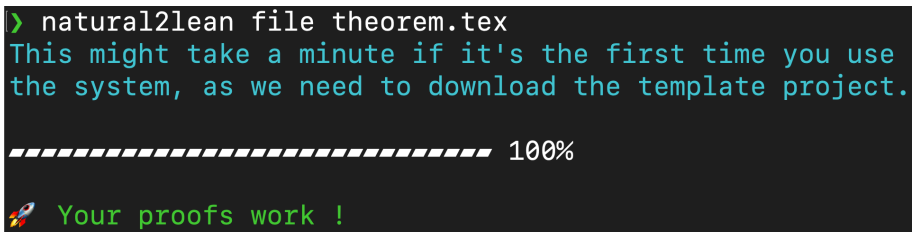

Figure 3.5: Full shortcut specifying an input file

Interactive mode

Using "`interactive`" or "`i`" will run the program directly in interactive mode (as shown in figure 3.6). "Enter a theorem statement" is the start of a proof and will be explained later (in section 3.2.6).

If too many or incorrect arguments are given to the program, it will print an error message and fallback to the command line interface (as shown in figure 3.7).

```
❯ natural2lean interactive
This might take a minute if it's the first time you use
the system, as we need to download the template project.

? Enter a theorem statement.
```

Figure 3.6: Shortcut for interactive mode

```
❯ natural2lean interactive theorem.tex
Should not specify input file in interactive mode.
? What mode do you want to use?
❯ interactive
  file
```

Figure 3.7: Incorrect command line arguments

3.2.4 Errors

There are three types of errors: translation errors, lean errors and conclusion errors. In the last two cases, the system will give feedback on how the sentence was understood to help you find the problem.

Translation errors

Translation errors arise when `natural2lean` was not able to understand a sentence (or part of it). In that case, check the keywords in section 3.1 to construct a sentence that can be parsed (see figure 3.8).

```
? Input a statement.
  $n$ can be expressed as $2m$

  The system could not understand your statement, try again.

Could not match any statement for '$n$ can be expressed as $2m$',
tried InductionCase, Induction, Have, ProofStructure
```

Figure 3.8: Translation error: impossible to parse

Lean errors

These errors come directly from the software that verifies statements. They can be caused by a wrong input (e.g., an error in a math expansion), by lean's limitations (this software is still under development and some correct statements cannot yet be verified), or by a non-optimal translation (some statements could be proved in lean,

but our translation was insufficient). An example is shown in figure 3.9. If you get a `lean error`, verify that your statement is correct, or check section 5.1, as some of lean's current limitations are explained there.

```
? Input a statement.
  We can show that $2+1 = 4$

We can show that $2+1 = 4$

 Lean could not assert that your statement is correct, try verifying
 it or take smaller steps.

Lean detected an error in your statement.
```

Figure 3.9: Lean error: $2 + 1 \neq 4$

Conclusion errors

Conclusion errors occur when the user input was matched as an attempt to finish a proof (or part of it), but at this point, lean could not finish the proof (see example in figure 3.10). In this case, try adding a step to get closer to the goal.

```
? Input a statement.
  It is obvious that $n^2$ is even.

It is obvious that $n^2$ is even .

 The system could not conclude a goal at this point, nor match a non
-conclusive statement in your input.

Could not match a non-conclusive statement, nor conclude a proof with
'It is obvious that $n^2$ is even.'.
```

Figure 3.10: Conclusion: tried to finish the proof too soon

3.2.5 File mode

Choosing the file mode allows the user to input a file line-by-line in the system. It skips blanks and commented lines (lines starting with "%"), but will still input inline comments. One example of an input file is shown in figure 3.11, but more can be found on GitHub[7].

A progress bar is displayed during the execution of the program (shown in figure 3.12), and a confirmation message is printed if the proofs could be verified (shown

[7] `natural2lean-cli/tests/examples`

```
% theorem statement
Theorem Square of even number is even: if $m \in \mathbb{N}$
is even, then $m^2$ is even.

% proof
By definition, if $m$ is even, we have a natural number $n$
such that $m = 2n$.

From that, we can derive $m^2 = (2n)^2 = 4n^2 = 2(2n^2)$

Hence, $m^2$ is even.
```

Figure 3.11: Input file for a simple proof

in figure 3.13). Note that writing multiple proofs in a single file will test each proof successively.

```
? What mode do you want to use? file
? Which file do you want to use? correct-theorem.tex

This might take a minute if it's the first time you
use the system, as we need to download the template
project.

▰▰▰▰▰▰▰▰▱▱▱▱▱▱▱▱▱▱▱▱▱▱▱▱▱▱▱▱▱▱ 28%
```

Figure 3.12: Progress indicator

If a line causes an error, the system will indicate which line caused the error along with how this line was matched, an indication on why the error occurred and the state of the system right before the error. An example of such feedback is given in figure 3.14.

3.2.6 Interactive mode

Input

After launching the interactive mode, the user is asked to enter a theorem statement. This one can be either named or unnamed (the format is explained in section 3.1.3). Progress is then made by entering statements (format explained in section 3.1.4) until nothing is left to prove. At this point, the system will ask the user to input

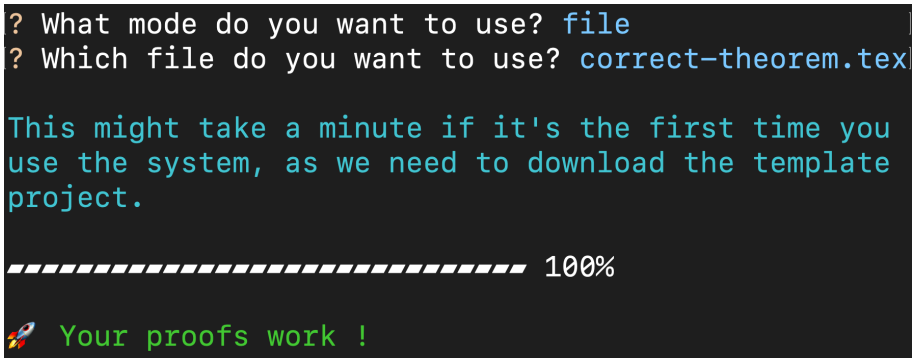

Figure 3.13: Success message

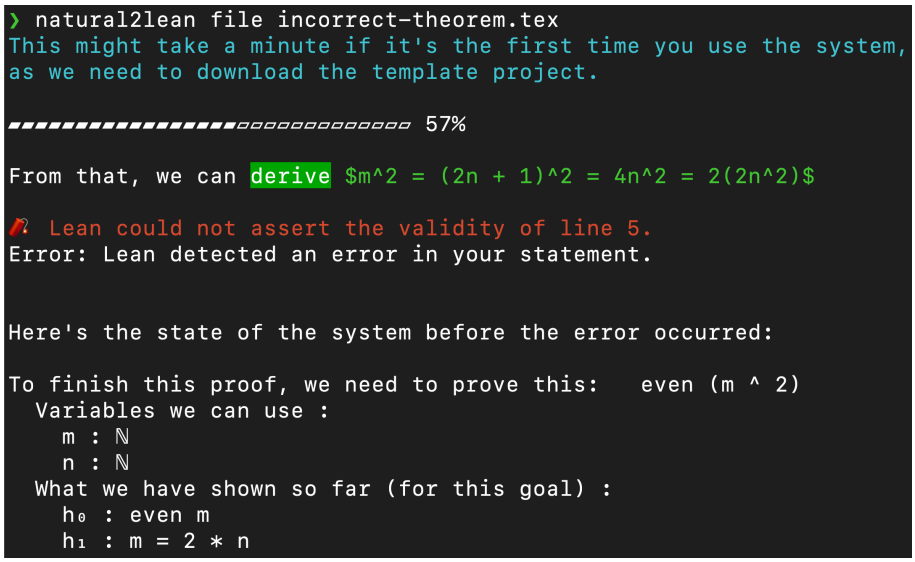

Figure 3.14: Error feedback

another theorem statement. A complete example of this is given in figure 3.15, and a step-by-step decomposition is given in section 3.3.3.

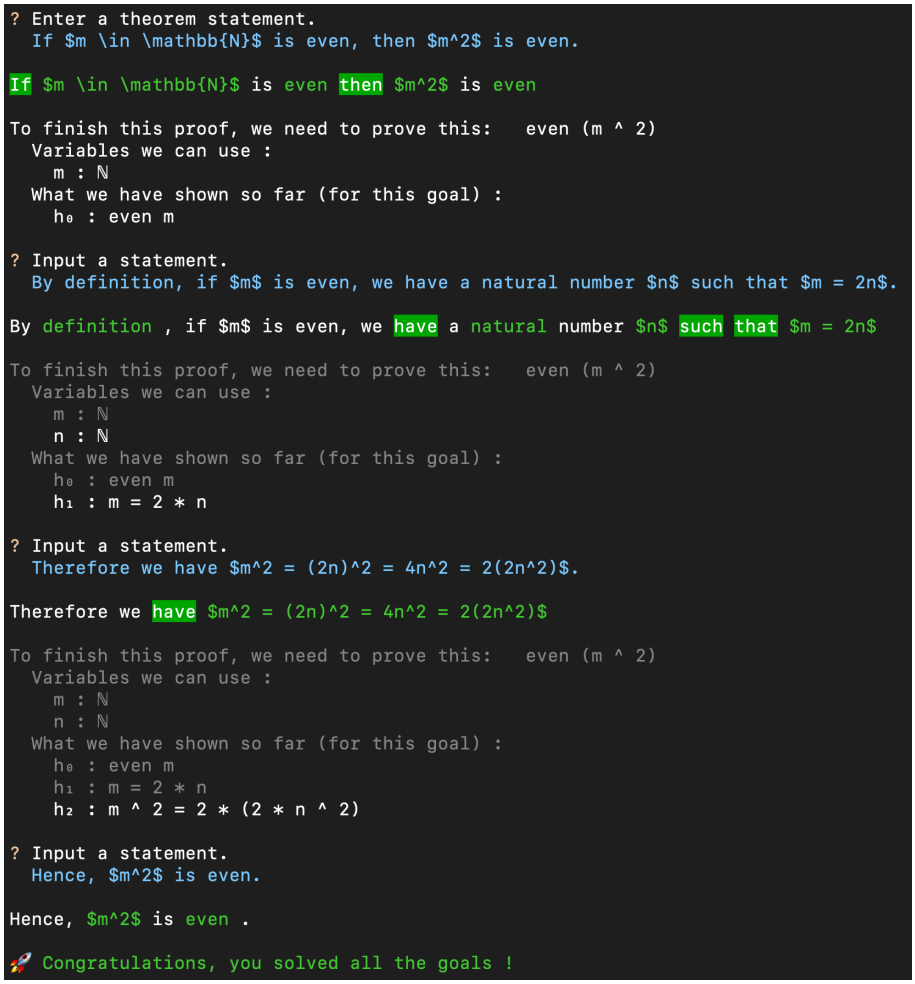

Figure 3.15: Complete proof in the CLI

At any point, one can input “back” or “backtrack” to cancel the last input given, and “exit” or “quit” to exit the program (note that your proofs will not be saved). Examples are shown in figure 3.16.

Feedback

When the input was verified as correct, the feedback given in interactive mode consists in two parts: the interpretation feedback, and the current state of the proof. If

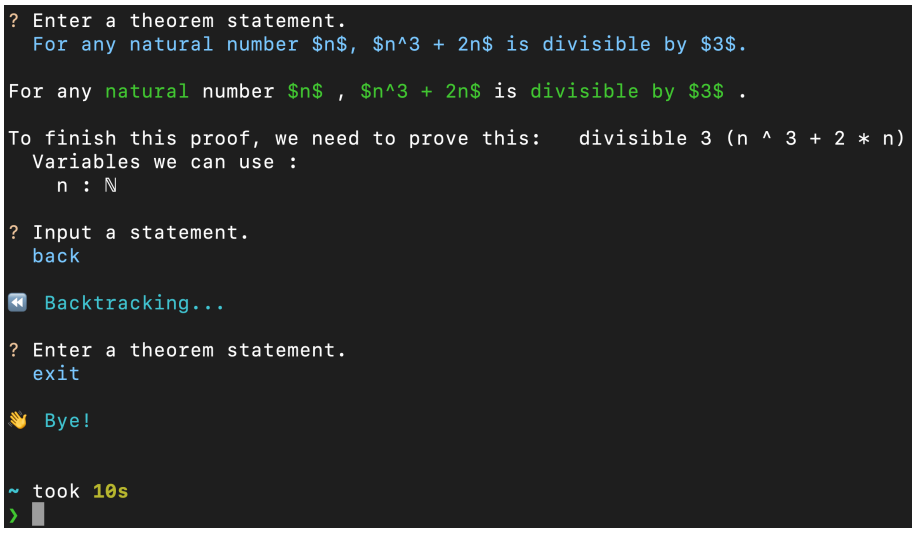

Figure 3.16: Backtrack and exit

there was an error, the feedback will indicate which type of error occurred (see section 3.2.4 for more information).

The interpretation feedback is an indication of how the input sentence was understood, highlighting keywords and coloring parameters. An example is shown in figure 3.17.

Figure 3.17: Interpretation feedback

The state of the proof consists of one or more blocks (one for each part left to prove) indicating what has to be shown, the declared variables along with their type, and the different statements that were already proven ("what we have shown so far"). As a big part of this information is repeated at every step, the lines that were already present in the previous state are greyed out. Figures 3.15 and 3.18 show examples of this.

Note that this feedback is parsed back from lean, but not translated back. Therefore, "even `m`" has to be understood as "m is even", "odd `m`" as "m is odd", and "`divisible 3 q`" as "q is divisible by 3". Negation is denoted with the symbol "¬". A translation of the feedback can be implemented in the future, but as this is only a small aspect of the program and only concerns "even", "odd" and "divisible by", we decided that spending time on the actual translation and other features was more important.

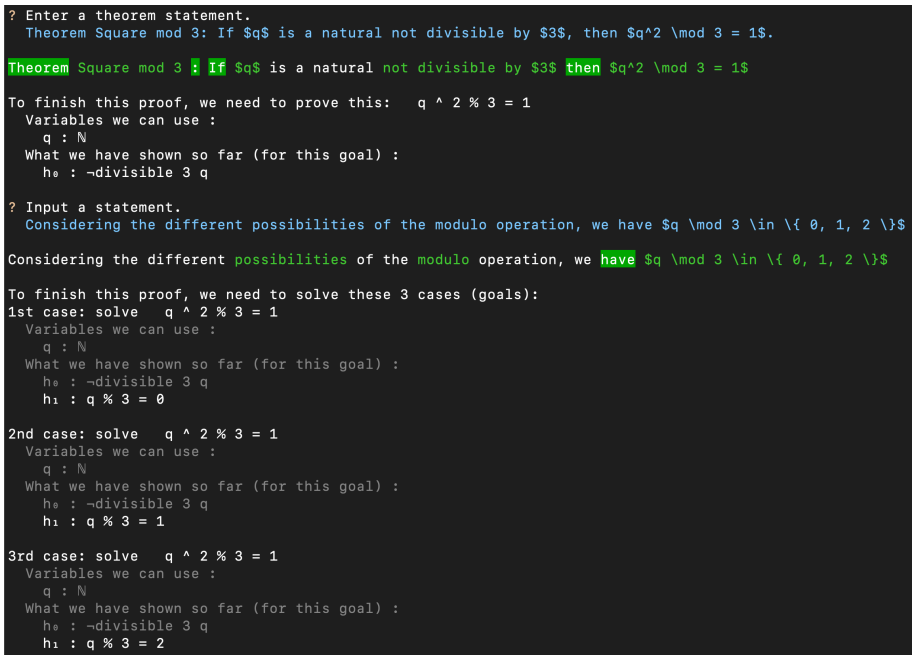

Figure 3.18: Feedback when splitting a proof

3.3 Working examples

3.3.1 Main examples

The four main examples used when developing the system are showed in examples 3.20, 3.21, 3.22 and 3.16, with keywords and parameters in bold. These can also be found on GitHub[8] and are inspired by [18]. More examples are given in appendix A[9] (examples A.1, A.2, A.3, and A.4). The first one is a simple proof, stating that if a number is even, its square is also even.

The second one is a bit longer. It states that if a number is not divisible by 3, then its square $\mod 3$ is equal to 1. It involves dividing into three cases and expanding each case to show the theorem is true. The full proof is given in example 3.21.

The third example (3.22) proves the same theorem as the previous one and proceeds by proving a contrapositive. To work correctly, we need to provide example 3.21 with a theorem name ("square mod 3" in this case) at the beginning of the file (if using file mode) or start by proving it (when using the interactive mode).

[8] `natural2lean-cli/tests/examples`
[9] Or on Github (`natural2lean-cli/tests/examples/variations`)

Theorem statement	**If $m \in \mathbb{N}$** is **even**, **then m^2** is **even**.
Proof	By **definition**, if **m** is **even**, we **have** a **natural** number **n such that $m = 2n$**. Therefore we **have $m^2 = (2n)^2 = 4n^2 = 2(2n^2)$**. Hence, **$m^2$** is **even**.

Example 3.20: If m is even, m^2 is too (direct proof)

Theorem statement	**If q** is a **natural not divisible by 3**, **then $q^2 \mod 3 = 1$**.
Splitting	Considering the different **possibilities** of the **modulo** operation, we **have $q \mod 3 \in \{ 0, 1, 2 \}**$.
1st case	For the first case, as q is not divisible by 3, we **have** a **contradiction**.
2nd case	For the second case, we **have $q \mod 3 = 1$**. By **definition** of the modulo operation, we **have $k \in \mathbb{N}$ such that $q = 3k + 1$**. By expansion, we **have $q^2 = (3k+1)^2 = 9k^2+6k+1 = 3(3k^2 + 2k) + 1$**.
Conclusion	Hence, **$q^2 \mod 3 = 1$**.
3rd case	For the last case, by **definition** of the modulo operation, we **have $k \in \mathbb{N}$ such that $q = 3k + 2$**. Following the same expansion, we **have $q^2 = (3k+2)^2 = 3(3k^2+4k+1) +1$**.
Conclusion	Hence, **$q^2 \mod 3 = 1$** again.

Example 3.21: If q is not divisible by 3, then $q^2 \mod 3 = 1$

Theorem statement	**If $q \in \mathbb{N}$** and **q^2** is **divisible by 3**, **then q** is also **divisible by 3**.
Contrapositive	We will prove the **contrapositive**.
Using 3.21	By the "**square mod 3**" theorem, we **have $q^2 \mod 3 = 1$**.
Conclusion	Hence **q^2** is **not divisible by 3**.

Example 3.22: If q^2 is divisible by 3, then q is also divisible by 3

The last example consists of an induction to prove that $n^3 + 2n$ is always divisible by 3. It was already used as example for the induction keyword in section 3.1.4 (example 3.16).

3.3.2 Different approaches

To show that one single theorem can be proven in different ways, we've proven that if m is even, m^2 is also even[10] by direct proof (already shown in example 3.20) and by induction (example 3.23). In addition to that, a proof of the divisibility by 3 of $n^3 + 2n$ was constructed by induction (already shown in example 3.16), but this can also be proven by cases (example 3.24).

Theorem statement	**If $m \in \mathbb{N}$** is **even**, **then m^2** is **even**.
Induction	We will use **induction** on **m**.
Base cases	The first base case, **$m=0$** is **trivial**. The second base case, **$m=1$** is also **trivial**.
Start induction step	For **$m = k+2$**, based on the fact that $k+2$ is even, we can **show** that **k** is **even**.
Induction hypothesis	Hence, the **induction hypothesis gives** us that **k^2** is **even**.
Development	But we know that if k^2 is even, we **have** a **natural** number **l such that $k^2 = 2l$**.
Development	Combining this, we **show** that **$(k+2)^2 = k^2 + 4k + 4 = 2l + 4k + 4 = 2(l+2k+2)$**.
Conclusion	Therefore, **$(k+2)^2$** is **even**.

Example 3.23: Proof by induction

[10] Also found on GitHub (`natural2lean-cli/tests/examples/different-approaches-m-sqr`)

Theorem statement	For any **natural** number **n**, **$n^3 + 2n$** is **divisible by 3**.
Splitting	Considering the different **possibilities** of the **modulo** operation, we **have $n \mod 3 \in \{ 0, 1, 2 \}$**.
1st case	In the first case, we **have** a **natural** number **k such that $n = 3k$**. Using this, we can **show** that **$n^3 + 2n = 27k^3 + 6k = 3(9k^3+2k)$**.
Conclusion	Therefore, **$n^3 + 2n$** is **divisible by 3**.
2nd case	For this second case, we **have** a **natural** number **k such that $n = 3k + 1$**. By expansion, we can **show** that **$n^3 + 2n = 27k^3 + 27k^2 + 15k + 3 = 3(9k^3 + 9k^2 + 5k + 1)$**.
Conclusion	And we can therefore conclude that **$n^3 + 2n$** is also **divisible by 3** in this case.
3rd case	The last case is similar, we **have** a **natural** number **k such that $n = 3k + 2$**. By expansion, we can **show** that **$n^3 + 2n = 27k^3 + 54k^2 + 42k + 12 = 3(9k^3 + 18k^2 + 14k + 4)$**.
Conclusion	We can therefore conclude that **$n^3 + 2n$** is **divisible by 3** for any number.

Example 3.24: Proof by cases

3.3.3 Step-by-step in interactive mode

In this section, we will analyze in detail an example of a relatively long proof using the interactive mode. This begins with figure 3.19, where we input the theorem statement. The system matches the "theorem" and ":" keywords, along with the theorem name ("Square mod 3"). The theorem statement is then separated into hypothesis ("q is not divisible by 3") and thesis ("$q^2 \mod 3 = 1$"), creating the given feedback.

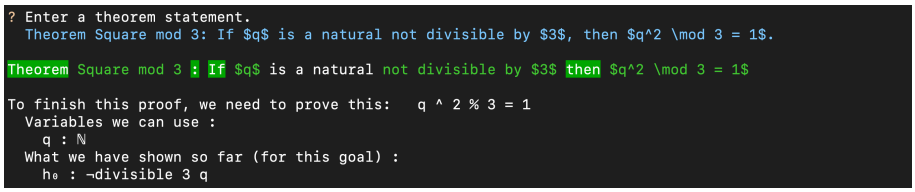

Figure 3.19: Theorem statement

The next step consists of separating the proof into 3 cases, for each possible value of $q \mod 3$. This is shown in figure 3.20, where we see that the keyword matched is have, and the system then finds the different values given for $q \mod 3$, along with the proof using the possibilities for the modulo operation.

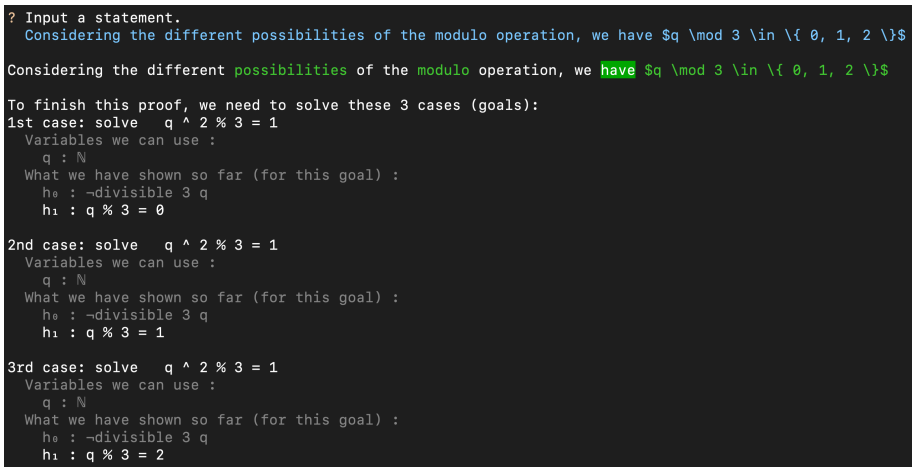

Figure 3.20: Separation into cases

We can then see in the feedback for the first case (in figure 3.20) that we have shown two contradicting statements: h_0 : ¬divisible 3 q and h_1 : q % 3 = 0. We can therefore state that we have a contradiction, which solves the 1st case. The feedback (shown in figure 3.21) then shows us that we solved a goal, and we only have 2 cases to solve.

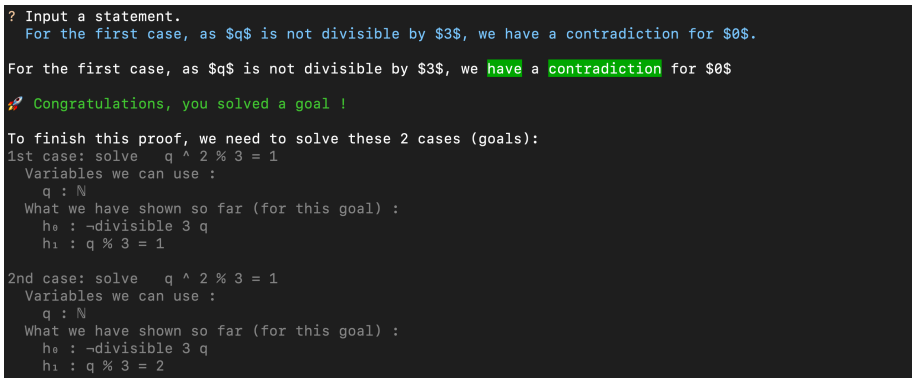

Figure 3.21: Solving the first case

This next statement (figure 3.22) is not necessary (as it produces exactly the same statement as what was already assumed in h_1), but if we are writing the proof in natural language, it can help clarify in which case we are[11].

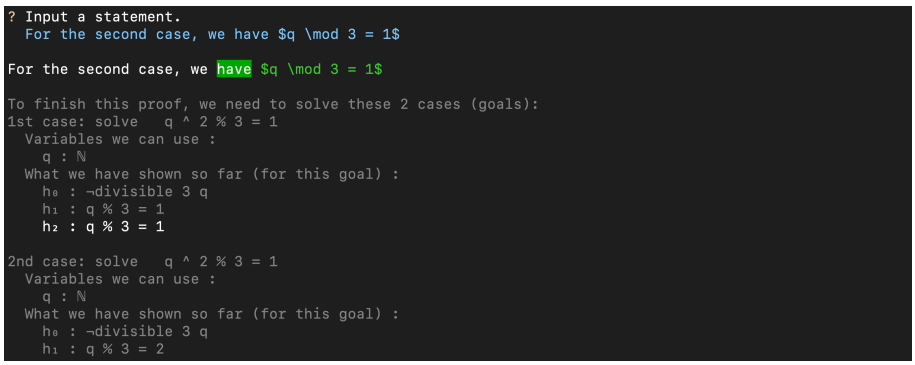

Figure 3.22: Starting the second case

We then create a variable k and show that $q = 3k + 1$ (in figure 3.23), derived directly from how the modulo operation is defined. This allows us to expand q^2 in figure 3.24, rewriting it in the form $3x + 1$.

Showing that q^2 can be written as $3x + 1$ allows us to conclude that $q^2 \mod 3 = 1$ (in figure 3.25). After this, the feedback tells us that we solved a goal and that there is one goal left. Showing us what we need to prove again, along with what has already been proven and the variables we can use.

[11] If we wanted to avoid this duplication, we could start this case with something like "`For the second case, since we know that $q \mod 3 = 1$, we` **`have $k \in \mathbb{N}$ such that $q = 3k + 1$`**", which would bring us directly to the state in figure 3.23.

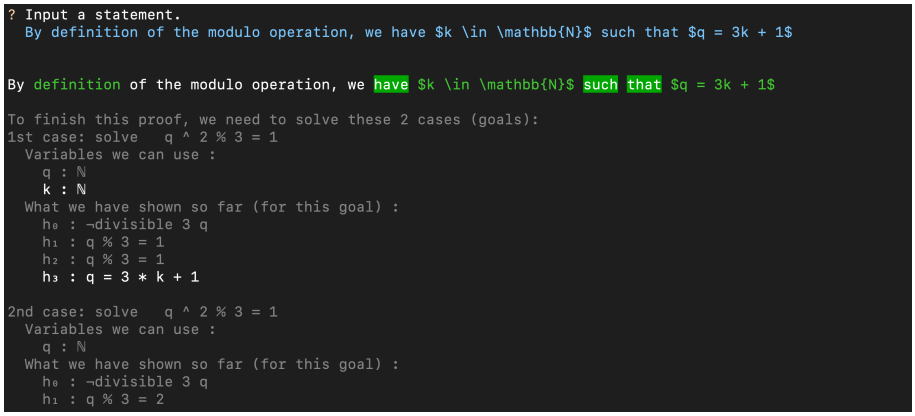

Figure 3.23: Introducing k

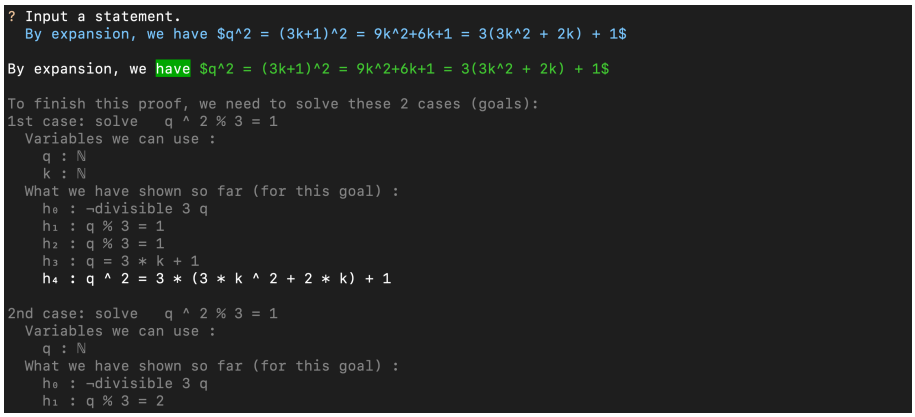

Figure 3.24: Expanding q^2 to the form $3x + 1$

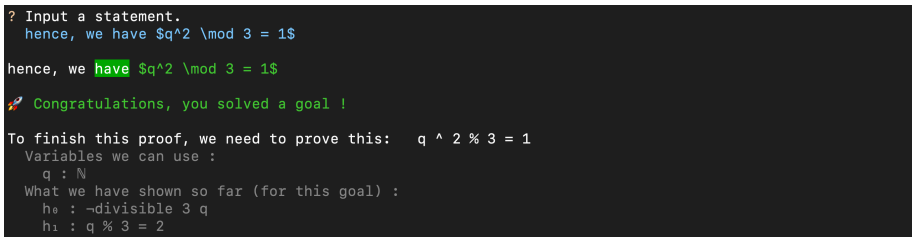

Figure 3.25: Concluding the 2nd case

To start the last case (in figure 3.26), we introduce k and the fact that $q = 3k + 2$. This is similar to what was done in figure 3.23 for the second case, and allows us to also expand q^2 again to the form $3x + 1$ (in figure 3.27). We then conclude the proof in figure 3.28 by simply stating that $q^2 \mod 3 = 1$.

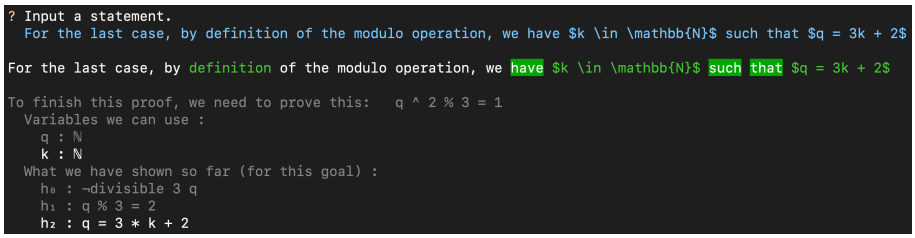

Figure 3.26: Starting the last case by introducing k

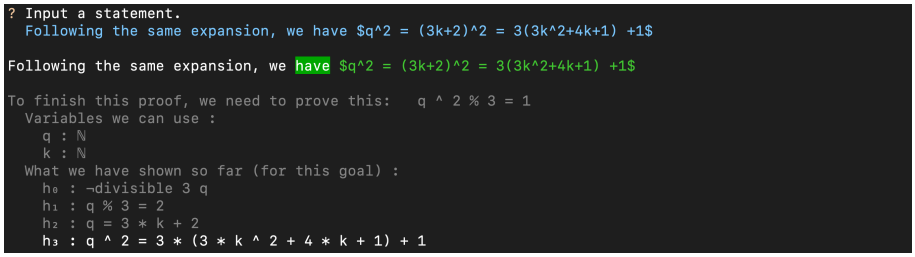

Figure 3.27: Expanding q^2 to the form $3x + 1$

Figure 3.28: Concluding the last case

3.4 Conclusion

The aim of this chapter was to explain the system to an end user, showing what can/can't be done and give him enough information to use the system. As shown in 3.2.6, a large amount of feedback is given to the user, notably the way the sentence was understood, which can help users get a better understanding of the different keywords that can be used and how to create sentences using them.

CHAPTER 4

Implementation

This chapter is aimed at a programmer willing to improve the system and describes how the program was implemented with more details, but knowledge of chapter 3 is recommended, as many references to it are made. Examples of translations of complete proofs are given in appendix B.

The main translation system is developed as a python package published on PyPi[1], and the command line interface is used as an example to show expandability. The latter is also published on PyPi[2].

4.1 Main program

4.1.1 Translator for interaction

The `Translator` class is used to interact with the translation system. Using this object's methods, we only have to think about the interaction with the user. This allowed us to create the command line interface in only 500 lines of code, managing the prompts for user input and the presentation of the feedback.

[1] `https://pypi.org/project/natural2lean/`
[2] `https://pypi.org/project/natural2lean-cli/`

Initializing

Upon creation of a `Translator` object, it will check if the project template[3] has already been cloned, and clone it if it wasn't. One can specify where to clone the project with the `lean_project_directory` argument if needed. This can take some time, as `mathlib4`[4] also needs to be cloned.

New input

After creating the instance, new theorems and statements can be added by calling the `new_theorem` or `new_statement` methods. Note that if there are goals left to prove, `new_theorem` will raise an error, and similarly, `new_statement` will raise an error when there are none, but the `new` method will call either one according to the current state of the `Translator`. All these methods take a string as argument (formatted in LaTeX, as explained in section 3.1) and return the new state of the system.

Backtrack

This simple method (`backtrack`) can be called to cancel the last valid input. It will return the state of the system after backtracking, or `None` if no input is left to cancel. This method is handled by `natural2lean`, and does not involve interacting with `lean`.

Interpretation

Two methods can be used to obtain feedback: `interpretation_feedback` and `failed_statement_interpretation`. They both return `lists` of `tuples` of type and section. The sections are parts of the string that can be reassembled to produce the input string, and the corresponding types ("`keyword`", "`parameter`" and "` ignored`") indicate how they were processed by `natural2lean` so that the software using it can give feedback to the user. Figure 3.17 in section 3.2.6 show one example of how this feedback can be given.

The `interpretation_feedback` method will return feedback on the last successful statement (or `None` if there was no input given), while the `failed_statement_interpretation` method will give feedback on the last failed statement (statement that raised a `LeanError` or a `NoConclusion` error, as statements raising `TranslationErrors` could not be understood), or `None` if no statement has failed yet. Failed statements are not affected by `backtrack`.

[3] `https://github.com/Augustindou/natural2lean-lean-project-template`
[4] `https://github.com/leanprover-community/mathlib4`

State

A state contains a list of LeanBlocks (goals that still need to be proven), the last statement given as input, as well as the complete translation of the input (in lean). Each of the LeanBlocks contains variables along with their set (a list of (name, set) tuples), hypotheses (a list of (name, hypothesis) tuples indicating what has already been proven or was assumed at the start) and a goal (what needs to be proven).

4.1.2 Structure

Translatable interface

To make sure that all our classes behave similarly, we created the Translatable interface that every class implements. This interface contains a translate method that returns a string translating the element in lean, as well as an interpretation_feedback method, that returns a list indicating how a specific element was understood. The similarly named methods mentioned in section 4.1.1 simply delegate to this one.

In addition to that, can_create_new_goals returns a boolean indicating whether a statement is allowed to split a goal or not. The reason for that is that when a statement is not complete, lean will most probably not throw an error, but create a new goal and ask for more development to accept that statement.

The change_status method also returns a boolean. It indicates whether the statement should change the proof's status, as solving a specific case for a variable has no sense if no induction has been started. It is specific to induction for now, but can be expanded.

When the constructor of a Translatable is called, it should check if it can parse the input string. If it can, it should create its sub-statements, but if it does not, it should raise a MatchingError (explained below). This allows the sub-statements to "cancel" the statement if they detect an error using error propagation.

Exceptions / Errors

MatchingErrors are used to indicate that a specific input cannot be parsed in a specific class, and therefore should be tried in other classes.

TranslationErrors occur when all the top-level classes raised a MatchingError for a specific input. If new_statement is called, but the input sentence doesn't match

any of the statements possibilities (i.e., they all return `MatchingErrors`), the system will raise a `TranslationError`.

`LeanErrors` are raised when an input could be parsed, but its translation caused an error in `lean`. This means that the input can be wrong (or parsed in a way that was not intended), or that the translation was not sufficient to prove the statement. In that case, the user should be advised to check their statement or create smaller ones.

The last type of error is `NoConclusion` and is raised when the system understood the input as the user wanting to close a goal, but could not succeed.

General structure of the files

The files are separated into 6 folders, with 4 of them corresponding to the different subsections used in section 3.1, one folder containing the code for interacting with lean (section 4.1.3, folder `lean_interaction`) and the utils folder, containing the exceptions, the `Translatable` interface, some text, and git utilities, along with a function to unfold multiple elements based on a regex pattern (used to separate identifiers or expressions and parse equations).

The `algebra` folder contains all the algebraic expressions detailed in section 3.1.1, `propositions` folder corresponds to section 3.1.2, while the different elements described in sections 3.1.3 and 3.1.4 are grouped into `proof_elements` (`theorem` and `statement` folders).

4.1.3 Interaction with lean

After a statement is matched, its `translate` method is called, and its output is concatenated to the translation of the previous statements. The whole translation is then written in the `Main.lean` file (in the project cloned when initializing the `Translator`). The project is then built, and the output is parsed into the feedback given to the user. Figure 4.1 is a visualization of this process.

This method of running a `build` at every input can obviously be improved, and it is the main cause of delay between input and feedback, but for the sake of this proof of concept, speed was not a concern, and we used this method as an easy way to get a prototype.

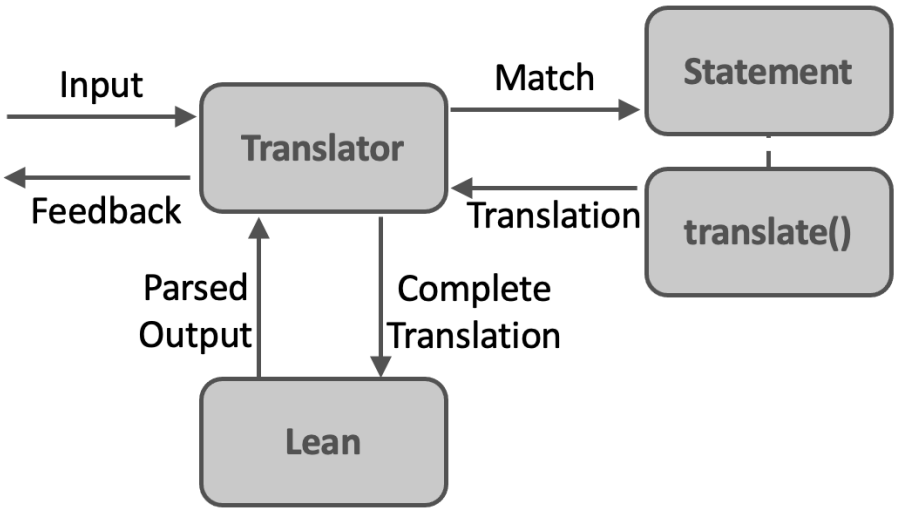

Figure 4.1: Interaction with `lean` from the `translator`

4.2 Command line interface

The command line interface is split between 3 files; `__main__.py`, `file.py` and `interactive.py`. The first one is the entry point of the CLI, it handles the first interaction with the user (checking for command line argument and asking the user for the mode and / or file if they were not given). After that, in file mode, the `file` function (in `file.py`) handles reading the input file and the interactions with the `Translator`. In interactive mode, the `interactive` function (in `interactive.py`) asks for user input and gives feedback (based on the response from the `Translator`).

4.3 Tests

This program has mainly been tested by exploration during development, but a bash script has been written to test all the working examples[5], essentially testing the whole program end-to-end.

4.4 Modularity & easy improvements

The system has been written with modularity in mind, i.e., most of the code relies on parameters that allows us to add functionalities (functions, keywords, ...) by only adding a few lines.

[5] `natural2lean-cli/tests`

Algebra translation

The translation of algebraic expressions to be understandable by lean (see section 3.1.1 for examples) is done by going through different parameters from a single file[6]. Adding items in this file allows us to easily add support for other LaTeX symbols, sets, and functions.

Example 4.1 shows the line needed for the system to parse integers correctly. The main part is a `MathSet` object containing a string with its expression in LaTeX, an intermediate representation used internally, `lean`'s name for the set and a tuple containing words that can then be used to describe the set in text (e.g., "`$x$ is an integer`"). One would then of course need to make sure that `lean` supports this set and adapt the existing `lean` functions to also work with it.

```
"ℤ": MathSet(r"\mathbb{Z}", "ℤ", "Int", ("integer",)),
```

Example 4.1: Needed line to add support for integers

Proofs in statements

With a small knowledge of lean, one can add keywords to improve translation of have[7] statements (mentioned in 3.1.4). For now, only "contradiction", "induction hypothesis", "definition" and possibilities for `mod 2` to `mod 5` are supported, but changing or adding new keywords only needs a pattern and a lean translation.

New proof structures[8] and simple statements[9] can also be added, and sub-statements can be changed easily for any statement.

Propositions

It is also easy to customize propositions thanks to the `proposition_constants`[10] file. This can be especially useful to add support for other functions (as for now, only even, odd and `divisible by` are supported). Example 4.2 shows what is needed for the odd function to work (`lean`'s name for the function, a regex pattern, and a callable to rearrange arguments). One should also keep in mind that this function has to be defined in `lean`.

[6] `natural2lean/algebra/translation_constants.py`
[7] `natural2lean/proof_elements/statements/have.py`
[8] `natural2lean/proof_elements/statements/proof_structure.py`
[9] `natural2lean/proof_elements/statements/simple_statements.py`
[10] `natural2lean/propositions/proposition_constants.py`

```
("odd", r"($.+?$).*?(odd)", keep_order),
```

Example 4.2: Needed line to support the "odd" keyword

CHAPTER 5

Limitations and possible improvements

We've seen what the system was capable of, but this is still a proof of concept. It is still quite limited by the use of the newer version of `lean`, but the modularity baked into the system allows for easy improvements. Other limitations and improvements are also discussed in this chapter.

5.1 `lean4` and `mathlib4`

One of the big limitations of the program is the current status of `lean4` and its main `mathlib4`[1] library. `Mathport`[2], a project to port the more powerful `mathlib3`[3] to `lean4` is currently in progress [19]. This would greatly improve the capabilities of `lean4`, but some work is still needed before the port.

Implementation of two proofs[4] was limited by the current state of `mathlib4`. Both of these are limited by the `ring` tactic and the first one[5] is also missing the `linearith` tactic (to solve inequalities)[6]. These two examples can be proved in `lean3`.

[1] `https://github.com/leanprover-community/mathlib4`
[2] `https://github.com/leanprover-community/mathport`
[3] `https://github.com/leanprover-community/mathlib`
[4] `natural2lean-lean-project-template/examples/not-working`
[5] `natural2lean-lean-project-template/examples/not-working/2-exp-n`
[6] As written by *Chris Bailey* in this discussion on the Zulip forum

5.2 General problem with parsing natural language

Using parsing to approach the auto-formalization problem requires the developers to envision every possible phrasing of proofs. This is not feasible for general natural to formal translations, and is the reason machine learning is the preferred approach in current research (as described in section 2.3).

However, it has the advantage of being explainable and deterministic and can therefore be useful in specific cases such as education. Following the rules explained in section 3.1 will guarantee a specific output, and the translation can be explained (as shown in section 3.2.6).

5.3 Improvements

5.3.1 Lean

One of the obvious ways to improve the system is to help with the development of `lean4` and `mathlib4`. Creating tactics that can solve what can seem obvious to a user or that translate a specific type of argument can then help create translations easily. We will then be able to improve `natural2lean` thanks to the modularity of our system. This has been described in section 4.4.

5.3.2 Tailoring to a specific course

The best way to make a parser work is probably to adjust it to a specific course, giving examples of correct and incorrect proofs, along with a good explanation of what is asked from the students in terms of rigor and development. Creating a general system would probably not work, as what can seem obvious to a master's student should probably be developed into multiple steps by a younger student. For example, proving that m^2 is even when m is even can be a good exercise for a high school student, but a master's student should be able to use that result without needing to prove it.

5.3.3 Gathering data

One of the limitations of machine learning techniques is the lack of data, but they could be used to create a system for teaching students how to prove theorems. To do that, teachers would first need to gather data, for example by asking students to prove theorems in LaTeX (or other desired input format) as an exercise, before labelling all statements as correct or incorrect, and proofs as correct, incorrect or incomplete.

The proof would then need to be written in a formal language, possibly using multiple structures and matching them to the specific proofs, as trying to translate a natural language proof by induction into a formal direct proof would not be useful. This would then allow us to train a model that would be able to translate proofs automatically.

Note that most of the current work in auto-formalization consists of trying to translate correct statements. Given enough data, such a program could therefore compensate a lack of rigor from a student with its knowledge of `lean` and would probably not be sufficient to detect small errors reliably. This is due to the fact that we cannot evaluate correctness of a proof using a model created with only correct data, as it is not the same task.

5.3.4 Hybrid approach

A hybrid approach could be envisioned, using machine learning to identify rules and patterns that could then be used to build a parser, but this also needs a lot of data. We could also imagine using AI to identify common patterns based on informal proofs before translating these by hand.

5.4 From natural language to formal proving

Translating natural language proofs into formal ones is useful to teach students about structure and give them a general idea on how to prove theorems, but to allow them to use the full power of a proof assistant and fine-tune their proofs, teaching them how to use these programs will give them a whole other set of tools.

This can then allow students and mathematicians to catch mistakes early, avoid publishing incorrect papers, and improve the state of mathematics. As explained by *Kevin Buzzard* at the International Congress of Mathematicians, formalization can also help simplify proofs, and even teach their authors how a proof works[7].

[7] In "The rise of formalism in mathematics" at 37:40, when explaining the formalization of the *fundamental theorem of liquid vector spaces* (proven by *Clausen* and *Scholze*)

CHAPTER 6

Conclusion

We wanted to prove that it is possible to give automatic feedback to students learning how to prove theorems. This was showed to be important to improve their ability to learn by giving more feedback than what would be feasible without automation.

We then discussed different approaches (machine learning and parsing) with their advantages and disadvantages when applied to the specific problems we set out to solve and elected the use of parsing.

We managed to create a system translating natural language sentences into formal code that is then verified by `lean4`, along with an interactive interface giving feedback at each sentence. This makes verifying simple proofs easier and allows the user to guarantee correctness without knowledge of formal theorem proving that can be complicated to master.

Our work was shown to be expandable and modular, as well as allowing for many ways to write proofs. It is easy to install, as it is published on PyPi, but it is also limited by the current state of the technologies and programs used. We proposed several improvements and showed that the use of a parser for auto-formalization was not the best option outside of education, and that it had to be specific.

We ended up by discussing the interest of teaching formal proofs and how it can help students and mathematicians to catch mistakes and get a deeper understanding of their proofs.

APPENDIX A

Variations on examples

Theorem statement	**If $m \in \mathbb{N}$** is **even**, **then $m^2 + 1$** is **odd**.
Proof	By **definition**, if m is even, we **have** a **natural** number **n such that $m = 2n$**. Therefore we **have $m^2 + 1 = (2n)^2 + 1 = 4n^2 + 1 = 2 (2n^2) + 1$** Hence, **$m^2 + 1$** is **odd**.

Example A.1: Simple theorem going from an even number to an odd number

Theorem statement	**If $m \in \mathbb{N}$** is **odd**, **then $m^3 + 1$** is **even**.
Proof	By **definition**, if m is odd, we **have** a **natural** number **n such that $m = 2n + 1$**. From that, we can **derive $m^3 + 1 = (2n + 1)^3 + 1 = 8n ^ 3 + 12 n ^ 2 + 6 n + 2 = 2 (4n^3 + 6n^2 + 3n + 1)$** Hence, **$m^3 + 1$** is **even**.

Example A.2: Simple theorem going from an odd number to an even number

Theorem statement	`For all` **`natural`** `numbers` **`$n$`**, **`$n^2+n$`** `is` **`even`**.
Induction	`We will use` **`induction`** `on` **`$n$`**.
Base case	`The base case (`**`$n=0$`**`) is` **`trivial`**.
Induction step	`For` **`$n=k+1$`**, `the` **`induction hypothesis gives`** `us that` **`$k^2+k$`** `is` **`even`**.
Expansion	`By expansion, we can` **`derive $(k+1)^2+(k+1) = k^2 + 3k + 2 = k^2 + k + 2(k+1)$`**.
Conclusion	`Therefore,` **`$k^2+k+2(k+1)$`** `is` **`even`**, `and $n^2+n$ is even.`

Example A.3: Induction to prove that $n^2 + n$ is even

Theorem statement	`For all` **`natural`** `numbers` **`$n$`**, **`$n(n+1)(n+5)$`** `is` **`divisible by $3$`**.
Induction	`We will use` **`induction`** `on` **`$n$`**.
Base case	`The base case (`**`$n=0$`**`) is` **`trivial`**.
Induction step	`For` **`$n=k+1$`**, `the` **`induction hypothesis gives`** `us that` **`$k(k+1)(k+5)$ is divisible by $3$`**.
Expansion	`By expansion, we` **`have $k(k+1)(k+5) = k^3 + 6k^2 + 5k$`**.
Expansion	`We can also` **`derive $(k+1)((k+1)+1)((k+1)+5) = (k+1)(k+2)(k+6) = k^3 + 9k^2 + 20k + 12$`**.
Splitting	`If we split this last expression, we can` **`show`** `that` **`$k^3 + 9k^2 + 20k + 12 = (k^3 + 6k^2 + 5k) + 3(k^2 + 5k + 4)$`**.
Conclusion	`Therefore` **`$k^3 + 9k^2 + 20k + 12$`** `is` **`divisible by $3$`**, `and thus $n(n+1)(n+5)$ is also divisible by $3$.`

Example A.4: Induction to prove that $n(n + 1)(n + 5)$ is divisible by 3

APPENDIX B

Examples of translated proofs

B.1 If m is even, m^2 is too

B.1.1 LaTeX

```
Theorem Square of even number is even: if $m \in \mathbb{N}$ is even,
then $m^2$ is even.

By definition, if $m$ is even, we have a natural number $n$ such that
$m = 2n$.

Therefore we have $m^2 = (2n)^2 = 4n^2 = 2(2n^2)$.

Hence, $m^2$ is even.
```

B.1.2 lean

```
import LeanUtils -- concepts such as even, odd, ...
open Nat

theorem square_of_even_number_is_even (m : Nat) (h₀ : even (m)) : even
    (m^2) := by

  have ⟨n, h₁⟩ : ∃ (n : Nat), m = 2*n := by simp at *; assumption
```

```
have h₂ : m^2 = 2*(2*n^2) := by
  calc
    m^2 = (2*n)^2 := by repeat (first | ring | simp_all)
    _ = 4*n^2 := by repeat (first | ring | simp_all)
    _ = 2*(2*n^2) := by repeat (first | ring | simp_all)

simp_all
```

B.2 If q is not divisible by 3, then $q^2 \% 3 = 1$

B.2.1 LaTeX

Theorem Square mod 3: **If q** is a **natural not divisible by 3**, **then $q^2 \mod 3 = 1$**.

Considering the different **possibilities** of the **modulo** operation, we **have $q \mod 3 \in \{ 0, 1, 2 \}$**.

For the first case, as q is not divisible by 3, we **have** a **contradiction**.

For the second case, we **have $q \mod 3 = 1$**.

By **definition** of the modulo operation, we **have $k \in \mathbb{N}$ such that $q = 3k + 1$**.

By expansion, we **have $q^2 = (3k+1)^2 = 9k^2+6k+1 = 3(3k^2 + 2k) + 1$**.

Hence, **$q^2 \mod 3 = 1$**.

For the last case, by **definition** of the modulo operation, we **have $k \in \mathbb{N}$ such that $q = 3k + 2$**.

Following the same expansion, we **have $q^2 = (3k+2)^2 = 3(3k^2+4k+1) +1$**.

Hence, **$q^2 \mod 3 = 1$** again.

B.2.2 lean

```
import LeanUtils
open Nat

theorem square_mod_3  (h₀ : ¬ divisible (3) (q)) : q^2 % 3 = 1 := by

  have h₁ : q % 3 = 0 ∨ q % 3 = 1 ∨ q % 3 = 2 := by (first | exact
    mod_2_poss _ | exact mod_3_poss _ | exact mod_4_poss _ | exact
    mod_5_poss _)
  rcases h₁ with h₁ | h₁ | h₁

  contradiction

  have h₂ : q % 3 = 1 := by
    calc
      q % 3 = 1 := by repeat (first | ring | simp_all)

  have ⟨k, h₃⟩ : ∃ (k : Nat), q = 3*k + 1 := by simp at *; assumption

  have h₄ : q^2 = 3*(3*k^2 + 2*k) + 1 := by
    calc
      q^2 = (3*k+1)^2 := by repeat (first | ring | simp_all)
      _ = 9*k^2+6*k+1 := by repeat (first | ring | simp_all)
      _ = 3*(3*k^2 + 2*k) + 1 := by repeat (first | ring | simp_all)

  apply mod_rewrite.mpr; exact ⟨_, by repeat (first | trivial | ring |
    simp_all)⟩

  have ⟨k, h₂⟩ : ∃ (k : Nat), q = 3*k + 2 := by simp at *; assumption

  have h₃ : q^2 = 3*(3*k^2+4*k+1) +1 := by
    calc
      q^2 = (3*k+2)^2 := by repeat (first | ring | simp_all)
      _ = 3*(3*k^2+4*k+1) +1 := by repeat (first | ring | simp_all)

  apply mod_rewrite.mpr; exact ⟨_, by repeat (first | trivial | ring |
    simp_all)⟩
```

Bibliography

[1] Muhammad Humayoun and Phd Defense. Developing the system mathnat for automatic formalization of mathematical texts. Technical report, University of Grenoble, 2012.

[2] Andres Zarza Davila. Proof of concept of an interactive theorem prover system using natural language input, 2021. Ecole Polytechnique de Louvain, Université Catholique de Louvain. Prom. : François Glineur.

[3] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[4] Achim D Brucker and Burkhart Wolff. Isabelle/dof: design and implementation. In *International Conference on Software Engineering and Formal Methods*, pages 275–292. Springer, 2019.

[5] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pak, and Josef Urban. Mizar: State-of-the-art and beyond. In *International Conference on Intelligent Computer Mathematics*, pages 261–279. Springer, 2015.

[6] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[7] Grzegorz Bancerek and Patricia Carlson. Mizar and the machine translation of mathematics documents, 11 1994.

[8] Grzegorz Bancerek, Adam Naumowicz, and Josef Urban. System description: Xsl-based translator of mizar to latex. In Florian Rabe, William M. Farmer, Grant O. Passmore, and Abdou Youssef, editors, *Intelligent Computer Mathematics*, pages 1–6, Cham, 2018. Springer International Publishing.

[9] Qingxiang Wang, Cezary Kaliszyk, and Josef Urban. First experiments with neural translation of informal to formal mathematics, 2018.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[11] Adam Naumowicz and Artur Korniłowicz. A brief overview of mizar. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 67–72, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[12] Google Cloud Translation - Evaluating models. `https://cloud.google.com/translate/automl/docs/evaluate#bleu`. Accessed: 2022-07-13.

[13] Qingxiang Wang, Chad Brown, Cezary Kaliszyk, and Josef Urban. Exploration of neural machine translation in autoformalization of mathematics in mizar. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, jan 2020.

[14] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

[15] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models, 2022.

[16] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

[17] A chat interface for formalizing theorem statements. https://leanprover.zulipchat.com/#narrow/stream/219941-Machine-Learning-for-Theorem-Proving/topic/A.20chat.20interface.20for.20formalizing.20theorem.20statements. Accessed: 2022-07-14.

[18] Lisa Oberbroeckling. Basic proof examples. http://math.loyola.edu/~loberbro/ma421/BasicProofs.pdf.

[19] Scott Morrison. Update on mathport. https://leanprover-community.github.io/blog/posts/intro-to-mathport/.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**