Waterproof: educational software for learning how to write mathematical proofs

Citation for published version (APA):
Wemmenhove, J., Beurskens, T., McCarren, S., Moraal, J., Tuin, D., & Portegies, J. (2022). Waterproof: educational software for learning how to write mathematical proofs. *arXiv*, *2022*, [2211.13513]. https://doi.org/10.48550/arXiv.2211.13513

DOI:
10.48550/arXiv.2211.13513

Document status and date:
Published: 24/11/2022

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Waterproof: educational software for learning how to write mathematical proofs

Jelle Wemmenhove, Thijs Beurskens, Sean McCarren,
Jan Moraal, David Tuin, Jim Portegies

November 2022

Abstract

In order to help students learn how to write mathematical proofs, we developed the educational software called *Waterproof* [1]. Waterproof is based on the Coq proof assistant. As students type out their proofs in the program, it checks the logical soundness of each proof step and provides additional guiding feedback. Contrary to Coq proofs, proofs written in Waterproof are similar in style to handwritten ones: proof steps are denoted using controlled natural language, the structure of proofs is made explicit by enforced signposting, and chains of inequalities can be used to prove larger estimates. To achieve this, we developed the Coq library *coq-waterproof* [2]. The library extends Coq's default tactics using the Ltac2 tactic language. We include many code snippets in this article to increase the number of available Ltac2 examples. Waterproof has been used to supplement teaching the course Analysis 1 at the TU/e for a couple of years. Students started using Waterproof's controlled formulations of proof steps in their handwritten proofs as well; the explicit phrasing of these sentences helps to clarify the logical structure of their arguments.

1 Introduction

At Eindhoven University of Technology (TU/e), many first-year students struggle with writing mathematical proofs. The main issue is not even the puzzle at the core of an exercise, it is more basic: at various proof stages, students are uncertain what is required of them and what they are, or are not, allowed to do. Students often neglect to introduce a variable when the goal is to show a $\forall$-statement and they are quick to assign extra properties to variables obtained from $\exists$-statements. In ε-δ-proofs, students unconsciously swap the order of quantifiers: they might come up with a value of δ that depends on a point x which is to be introduced only after the choice of δ has been made. In some sense the struggle is necessary: true understanding requires engaging with the material and making mistakes. After practicing all semester, however, some students are still confused about the mechanics underlying mathematical proofs.

1.1 Proof assistants

One attempt to improve the learning process of proof writing is by using special computer programs called *proof assistants*, like Lean [dMKA+15] or Coq [Coq22]. Students type out their proofs in the particular syntax of these programs and the computer then checks the logical soundness of each proofstep. Because of the direct feedback, students can freely explore which actions are allowed at the different stages of a proof; mechanical steps, like introducing a variable when showing a $\forall$-statement, are reinforced until they become second nature.

As long as proof assistants have been around, researchers have tried to harness their power in education; such cases are more numerous in communities that actively use proof assistants themselves. The earliest example we could find is the use of Mizar [BBG+15] to teach propositional logic at the

[1] *Waterproof* can be found at github.com/impermeable/waterproof, this article pertains to Waterproof version 0.6.1. Some exercise sheets that can be completed using Waterproof can be found at github.com/impermeable/waterproof-exercise-sheets.

[2] *coq-waterproof* can be found at github.com/impermeable/coq-waterproof, this article pertains to coq-waterproof version 1.2.4.

University of Warsaw in the 70s, according to [RZ05]. In computer science, proof assistants are often used to teach more advanced topics like formal logic [Nip12, HH11] or type theory [MKvRW10, §3]. The incorporation of proof assistants makes these theoretical courses feel more practical, as writing a proof with a proof assistant resembles coding. As more mathematicians are starting to use proof assistants, they have also found their way into mathematics courses: Heather MacBeth and Patrick Massot both use Lean to teach analysis; they discussed their experiences on a panel *Teaching with proof assistants* at the 2021 meeting of the Lean community [BAN+21]. Proof assistants are not solely used to more effectively teach a course's content: the account by Nipkow [Nip12] emphasizes that they used Isabelle [NWP02] specifically with the intention to improve the quality of students' pen-and-paper proofs.

Despite their benefits, we find proof assistants to be lacking for the purpose of education. These programs were never designed for the classroom, but for efficient use by experts. The following is a list of issues that come with using proof assistants 'out-of-the-box' for teaching mathematical proof writing:

P1. The ability to write proofs in a proof assistant does not necessarily transfer to an ability to write proofs with pen and paper. Although Thoma and Iannone [TI21] found that students who partook in a workshop on Lean produced better proofs, with more structure and correct use of mathematical language and symbols, a study by Knobelsdorf et al. [KFBK17] found that students in a small course involving Coq performed worse at writing proofs with pen and paper than with Coq. The members of the Lean panel attested to a similar statement: although students were able to master proof writing with the proof assistant, their handwritten proofs left much to be desired. Knobelsdorf et al. suggest that the transfer of proof skills might fail because Coq offers additional scaffolding that was not dismantled carefully during their course. In contrast to pen and paper, for example, proof assistants clearly display the current goal and hypotheses at all proof stages.

P2. It takes time to learn a proof assistant's syntax, see Figure 1 for the example of a Coq proof. Students would have to learn this syntax from scratch, whilst still not fully comfortable with the phrases used in ordinary proofs. Böhne and Kreitz [BK18] remark that Coq's syntax is difficult to learn: the *tactics* — key phrases that indicate proof steps, e.g. '`intro`' and '`destruct`' in Figure 1 — have unstructured names, do not distinguish explicitly in treatment of assumptions and goals, and may produce side effects (meaning that tactics can be applied in situations which you might not suspect from their description).

P3. The feedback that proof assistants give is limited to the soundness of isolated proof steps. This is not the only kind of feedback that students need: some students get stuck and need a hint in order to continue with a proof, others produce proofs that are technically correct, but too involved. Think of students neglecting to use lemmas and trying to derive everything from first principles.

P4. The installation of proof assistants is difficult. Most mathematics freshmen have never used the command line before, and students at the TU/e use personal laptops instead of fixed computers in a computer lab, so pre-installing a proof assistant is not an option.

P5. The user interfaces of proof assistants are uninviting. Take the CoqIDE for example: its visual design is slightly dated, giving the impression that the program has not seen maintenance in a while, which can lead users to doubt its reliability. Both Coq and Lean offer plugins for VS Code, but this editor was designed for programming and the user interface expresses this as well. Although computer science students may find comfort in what is to them a familiar environment, mathematics majors are used to seeing proofs on blackboards and in textbooks.

P6. Proof assistants do not allow for LaTeX-formatted expressions. Although proof assistants like Coq and Lean allow for Unicode notation, this is a large downgrade from the beauty and versatility of LaTeX. This is not a superficial requirement either: good notation clarifies mathematical concepts and aids understanding.

1.2 Waterproof

To address the issues above, we created *Waterproof*, a custom proof assistant based on Coq. By design, writing a proof in Waterproof closely resembles writing a proof by hand; we expect that closing this

```
Lemma example_coq :
  forall ε : R, ε > 0 -> exists a : R, ([0,4) a) /\ 4 - ε < a.
Proof.
  intro ε. intro ε_gt_0.
  assert (ε < 2 \/ 2 <= ε) as cases by lra.
  destruct cases as [ε_lt_two | two_le_ε].
  - (* Case ε < 2. *)
    exists (4 - ε/2).
    split.
    + split.
      * assert (0 <= 4 - ε/2) as h1 by lra; exact h1.
      * assert (4 - ε/2 < 4) as h2 by lra; exact h2.
    + assert (4 - ε < 4 - ε/2) as h3 by lra; exact h3.
  - (* Case ε ≥ 2. *)
    exists 3.
    split.
    + assert (0 <= 3 < 4) as h4 by lra; exact h4.
    + assert (4 - ε < 3) as h5 by lra; exact h5.
Qed.
```

Figure 1: Coq proof of the proposition $\forall\, \varepsilon > 0\, \exists\, a \in [0,4),\ 4 - \varepsilon < a$, part of the proof that $\sup[0,4) = 4$. (Advanced Coq users might be surprised by the length of the above proof, as the 'lra' tactic is powerful enough to finish each case directly after the 'exists' tactics; the proof here is more detailed to show the extent to which a pen-and-paper style proof can be imitated in default Coq.)

gap will help with transferring proof skills to pen and paper (issue P1) and make proof assistants more accessible for new users (issue P2). Waterproof consists of two components: a Coq library *coq-waterproof* which makes Coq proofs more similar to handwritten ones in terms of style, and a custom editor designed around the specific needs of teachers and students. The editor offers a solution to issues P4–P6, but it also aids a little with issues P1 and P2 by making the *act* of proof writing in Waterproof more similar to that with pen and paper.

Figure 2 shows an example of a proof written using the coq-waterproof library; it follows the same reasoning as the Coq proof in Figure 1, but resembles a handwritten proof more closely:

- As we require from students, the proof steps are denoted with full sentences instead of short keywords.

- The proof is signposted with statements like 'Case (ε < 2).'. Proofs written with regular proof assistants often forego signposting altogether, because the program constantly displays the current goal in a separate panel. With coq-waterproof, however, signposting is not just made possible, but enforced.

- Basic statements like $\varepsilon < 2 \,\vee\, \varepsilon \geq 2$ do not need to be justified. Such justifications are missing in Figure 2, whereas the default Coq proof does require these in the form of the text 'by lra'. Coq-waterproof uses an automation system to find these justifications on its own; we have attempted to tune the system such that it will only find proofs for 'trivial' statements, meaning proof steps for which we would not require further explanation from our students in Analysis 1.

- Coq-waterproof allows one to reason with chains of (in)equalities like $0 < 4 - 1 < 4 - \varepsilon/2 = a$ to show $0 < a$; although such chains are a key part of mathematical proofs, especially in analysis, Coq does not support this reasoning technique by default.

The screenshot in Figure 3 and 4 show the Waterproof editor's modern look, but there are more features that make the editor stand out:

```
Lemma example_coq_waterproof :
  for all ε : ℝ, ε > 0 ⇒ there exists a : ℝ, a : [0,4) ∧ 4 - ε < a.
Proof.
  Take ε : ℝ. Assume that (ε > 0).
  Either (ε < 2) or (ε ≥ 2).
  - Case (ε < 2).
    Choose a := (4 - ε/2).
    We show both (a : [0,4)) and (4 - ε < a).
    + We need to show that (0 ≤ a ∧ a < 4).
      We show both (0 ≤ a) and (a < 4).
      * We conclude that (& 0 < 4 - 1 < 4 - ε/2 = a).
      * We conclude that (a < 4).
    + We conclude that (4 - ε < a).
  - Case (ε ≥ 2).
    Choose a := 3.
    We show both (3 : [0,4)) and (4 - ε < 3).
    + We conclude that (3 : [0,4)).
    + We conclude that (& 4 - ε ≤ 4 - 2 = 2 < 3).
Qed.
```

Figure 2: Proof of $\forall\, \varepsilon > 0\, \exists\, a \in [0,4),\ 4 - \varepsilon < a$ using coq-waterproof, compare with Figure 1.

- Coq code — axioms, definitions, proofs — can be interspersed with formatted text, including LaTeX, to improve readability. These texts could explain the intuition underlying a proof, or describe an assignment that students have to complete using Waterproof; the same can be achieved in Coq with regular comments, but the standard way of rendering these in a nice formatting turns the document into a static file that can no longer be edited.

- Editing of a Waterproof file can be restricted to specific segments. When using Waterproof files as exercise sheets where some proofs are left to be completed, it is important that students are not able to (accidentally) change the parts of the file which specify the exercise.

- Typing mathematical symbols is made easy with a drop-down menu. For example, to write the ∞-symbol one starts typing '`\infty`' and, after the first couple of characters, a drop-down menu will appear with a list of mathematical symbols to choose from.

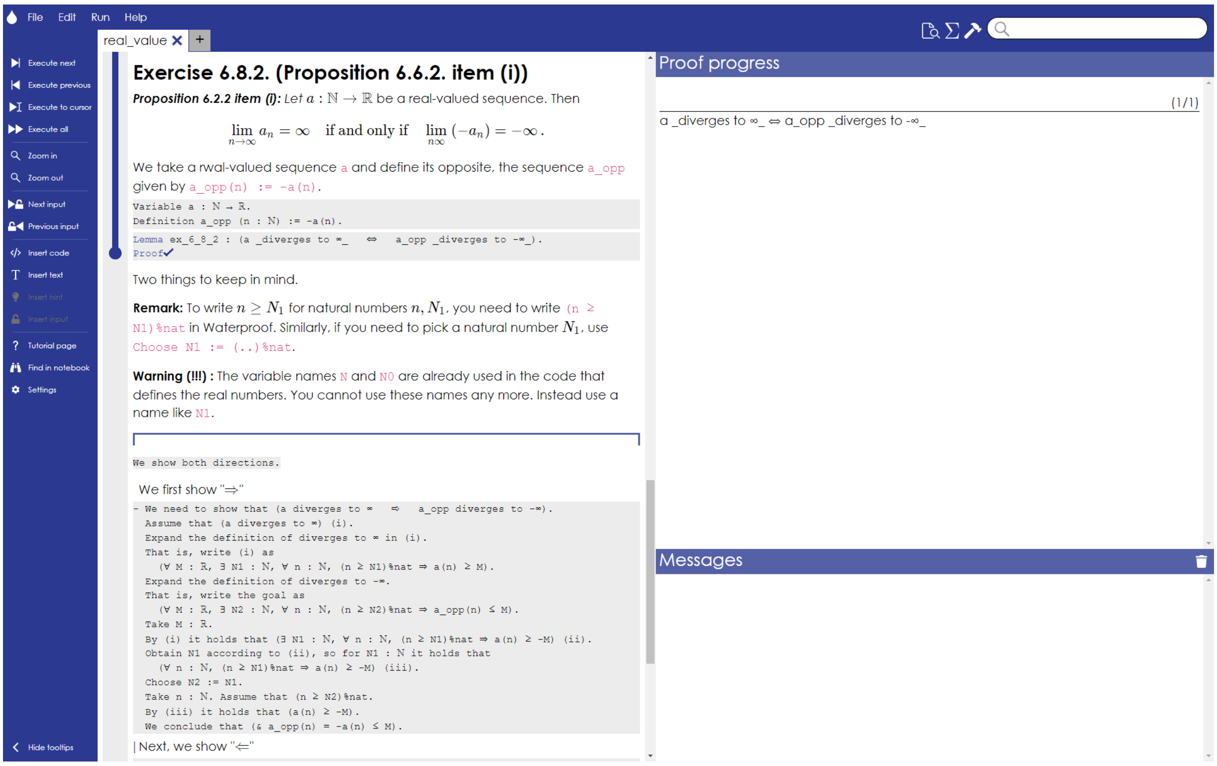
- An overview of the coq-waterproof tactics can be displayed on the side of the screen. The effect of these tactics is explained and they can be easily copied to the input area via buttons on the right.

- In contrast to other proof assistants, the current hypotheses are not explicitly displayed in a separate panel. This is a first attempt to remove features that are not present when writing a proof with pen and paper, but which students might have learned to rely on nonetheless.

For the last couple of years, Waterproof has been used to supplement teaching mathematical analysis at the TU/e. Roughly 175 students register for the course (Analysis 1, 5 ECTS), the majority are first-year undergraduate students. Students have to hand in weekly homework exercises in groups of 4, Waterproof exercise sheets have been created for a selection of assignments and these can be handed in instead of handwritten proofs.

Use of Waterproof has been optional and only a couple of instructors could answer questions about Waterproof, naturally limiting its adoption by students. At the start of the 2021/22 course, 16 student groups handed in homework written in Waterproof; 9 groups still used it for the final assignment.

Section 2 discusses alternative solutions to issues P1–P6 that come with using ordinary proof assistants in education, Section 3 argues why we based Waterproof on the Coq proof assistant specifically. The article then delves deeper into the inner workings of Waterproof: Section 4 details the design of the coq-waterproof library and Section 5 that of the editor. The section on coq-waterproof showcases many



a _diverges to ∞_ ⇔ a_opp _diverges to -∞_

Exercise 6.8.2. (Proposition 6.6.2. item (i))

Proposition 6.2.2 item (i): *Let $a : \mathbb{N} \to \mathbb{R}$ be a real-valued sequence. Then*

$$\lim_{n\to\infty} a_n = \infty \quad \text{if and only if} \quad \lim_{n\infty} (-a_n) = -\infty\,.$$

We take a rwal-valued sequence a and define its opposite, the sequence a_opp given by a_opp(n) := -a(n).

```
Variable a : N → R.
Definition a_opp (n : N) := -a(n).
```

```
Lemma ex_6_8_2 : (a _diverges to ∞_   ⇔   a_opp _diverges to -∞_).
Proof✓
```

Two things to keep in mind.

Remark: To write $n \geq N_1$ for natural numbers n, N_1, you need to write (n ≥ N1)%nat in Waterproof. Similarly, if you need to pick a natural number N_1, use Choose N1 := (..)%nat.

Warning (!!!) : The variable names N and N0 are already used in the code that defines the real numbers. You cannot use these names any more. Instead use a name like N1.

```
We show both directions.
```

We first show "⇒"

```
- We need to show that (a diverges to ∞   ⇒   a_opp diverges to -∞).
  Assume that (a diverges to ∞) (i).
  Expand the definition of diverges to ∞ in (i).
  That is, write (i) as
    (∀ M : R, ∃ N1 : N, ∀ n : N, (n ≥ N1)%nat ⇒ a(n) ≥ M).
  Expand the definition of diverges to -∞.
  That is, write the goal as
    (∀ M : R, ∃ N2 : N, ∀ n : N, (n ≥ N2)%nat ⇒ a_opp(n) ≤ M).
  Take M : R.
  By (i) it holds that (∃ N1 : N, ∀ n : N, (n ≥ N1)%nat ⇒ a(n) ≥ -M) (ii).
  Obtain N1 according to (ii), so for N1 : N it holds that
    (∀ n : N, (n ≥ N1)%nat ⇒ a(n) ≥ -M) (iii).
  Choose N2 := N1.
  Take n : N. Assume that (n ≥ N2)%nat.
  By (iii) it holds that (a(n) ≥ -M).
  We conclude that (& a_opp(n) = -a(n) ≤ M).
```

| Next, we show "⇐"

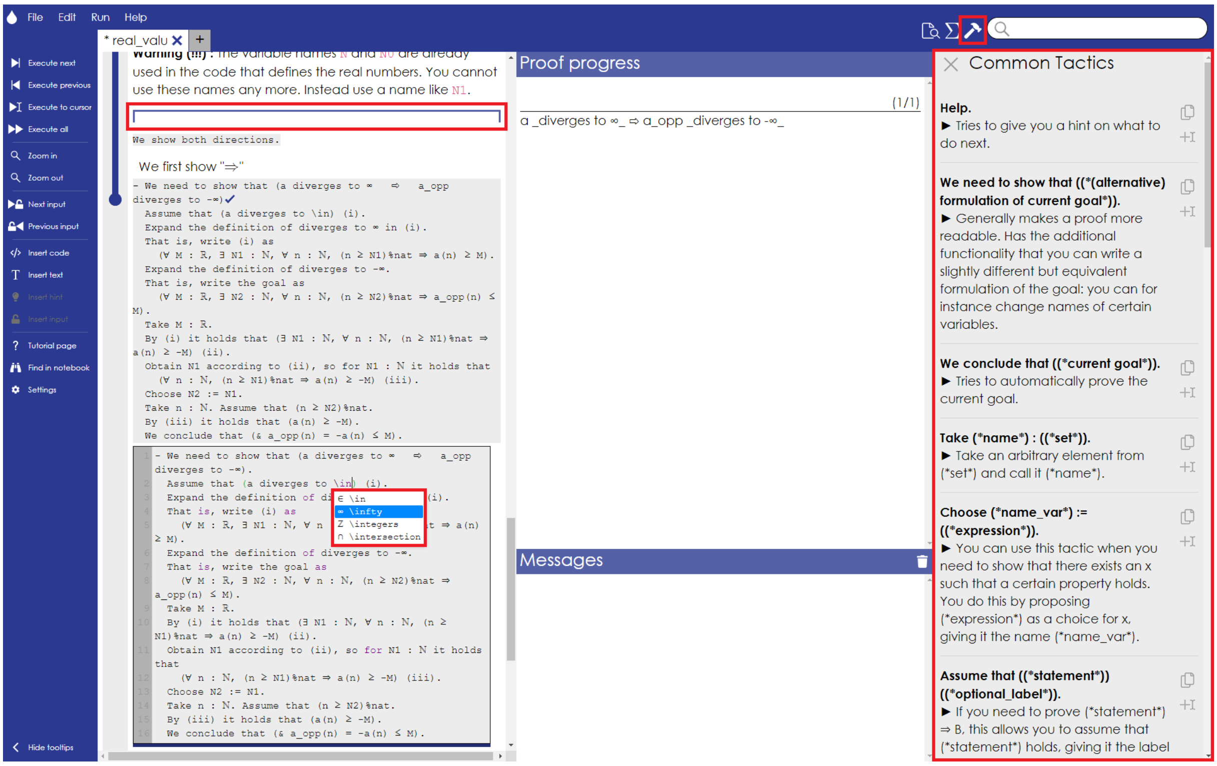
Figure 3: Screenshot of *Waterproof*. On the left is a file, on right are panels showing the current goal and error messages. A Waterproof file consists of formatted text (white background) and Coq code (gray background), the latter can be checked for logical correctness. The blue bar to the left of the file indicates which Coq sentences have been verified to be correct; the most recently verified one is followed by a check mark and indicates the current proof stage.

examples of the Ltac2 language for the creation of custom Coq tactics. We close off with a preliminary survey of students' opinions on Waterproof in Section 6, a discussion of future improvements in Section 7, and a conclusion in Section 8.

With this paper we hope to excite the collective imagination of what proof assistants could look like when designed for teaching how to write mathematical proofs.

Acknowledgements

In 2019, the first version of Waterproof arose as the result of a Software Engineering Project (SEP) at Eindhoven University of Technology, which is part of the bachelor curriculum. We would like to thank the members of the SEP team ChefCoq (2019): Thijs Beurskens, Bas Gieling, Stijn Gunter, Menno Hofsté, Hugo Melchers, Anne Nijsten, Reinier Schmiermann, Erik Takke, David Tuin, Ruben Verhaegh, Geert van Wordragen, and their supervisor Tom Verhoeff. In 2021, an iteration was made to Waterproof in another Software Engineering Project and we would like to thank the SEP team Waterfowl (2021): Adrien Castella, Adrian Cucoş, Cosmin Manea, Noah van der Meer, Lulof Pirée, Mihail Ţifrea, Tristan Trouwen, Tudor Voicu, Adrian Ş. Vrămuleţ, Yuqing Zeng, and their supervisor Gerard Zwaan. We would also like to thank Emilio Jesús Gallego Arias for his support, especially with coq-serapi, and his quick answers to many questions. We would like to thank Georgios Skantzaris for his feedback and help with testing when getting Waterproof ready for teaching for the first time. Finally we would like to thank EdIn (Education Innovation) at Eindhoven University of Technology for financing the work of the students Thijs Beurskens, Sean McCarren, Jan Moraal and David Tuin on this project.



Warning (!!!) : The variable names N and N0 are already used in the code that defines the real numbers. You cannot use these names any more. Instead use a name like N1.

```
We show both directions.
```

We first show "⇒"

```
- We need to show that (a diverges to ∞   ⇒   a_opp
diverges to -∞)✓
  Assume that (a diverges to \in) (i).
  Expand the definition of diverges to ∞ in (i).
  That is, write (i) as
    (∀ M : ℝ, ∃ N1 : ℕ, ∀ n : ℕ, (n ≥ N1)%nat ⇒ a(n) ≥ M).
  Expand the definition of diverges to -∞.
  That is, write the goal as
    (∀ M : ℝ, ∃ N2 : ℕ, ∀ n : ℕ, (n ≥ N2)%nat ⇒ a_opp(n) ≤
M).
  Take M : ℝ.
  By (i) it holds that (∃ N1 : ℕ, ∀ n : ℕ, (n ≥ N1)%nat ⇒
a(n) ≥ -M) (ii).
  Obtain N1 according to (ii), so for N1 : ℕ it holds that
    (∀ n : ℕ, (n ≥ N1)%nat ⇒ a(n) ≥ -M) (iii).
  Choose N2 := N1.
  Take n : ℕ. Assume that (n ≥ N2)%nat.
  By (iii) it holds that (a(n) ≥ -M).
  We conclude that (& a_opp(n) = -a(n) ≤ M).
```

```
- We need to show that (a diverges to ∞   ⇒   a_opp
diverges to -∞).
  Assume that (a diverges to \in) (i).
  Expand the definition of di                 (i).
  That is, write (i) as
    (∀ M : ℝ, ∃ N1 : ℕ, ∀ n                 t ⇒ a(n)
≥ M).
  Expand the definition of diverges to -∞.
  That is, write the goal as
    (∀ M : ℝ, ∃ N2 : ℕ, ∀ n : ℕ, (n ≥ N2)%nat ⇒
a_opp(n) ≤ M).
  Take M : ℝ.
  By (i) it holds that (∃ N1 : ℕ, ∀ n : ℕ, (n ≥
N1)%nat ⇒ a(n) ≥ -M) (ii).
  Obtain N1 according to (ii), so for N1 : ℕ it holds
that
    (∀ n : ℕ, (n ≥ N1)%nat ⇒ a(n) ≥ -M) (iii).
  Choose N2 := N1.
  Take n : ℕ. Assume that (n ≥ N2)%nat.
  By (iii) it holds that (a(n) ≥ -M).
  We conclude that (& a_opp(n) = -a(n) ≤ M).
```

```
∈ \in
∞ \infty
ℤ \integers
∩ \intersection
```

a _diverges to ∞_ ⇒ a_opp _diverges to -∞_

Help.
► Tries to give you a hint on what to do next.

We need to show that ((*(alternative) formulation of current goal*)).
► Generally makes a proof more readable. Has the additional functionality that you can write a slightly different but equivalent formulation of the goal: you can for instance change names of certain variables.

We conclude that ((*current goal*)).
► Tries to automatically prove the current goal.

Take (*name*) : ((*set*)).
► Take an arbitrary element from (*set*) and call it (*name*).

Choose (*name_var*) := ((*expression*)).
► You can use this tactic when you need to show that there exists an x such that a certain property holds. You do this by proposing (*expression*) as a choice for x, giving it the name (*name_var*).

Assume that ((*statement*)) ((*optional_label*)).
► If you need to prove (*statement*) ⇒ B, this allows you to assume that (*statement*) holds, giving it the label

Figure 4: Screenshot of Waterproof with some features highlighted in red. [1] A horizontal-[bracket which indicates the start of an input segment. Editing outside input segments can be restricted, such that Waterproof files can be used as exercise sheets with some proofs left to be completed. [2] A drop-down menu that aids with writing mathematical symbols like ∞. [3] A side-panel with an overview of coq-waterproof tactics, together with explanations and buttons for quickly copying them. Clicking the hammer-icon above the side panel controls whether the side-panel is shown.

2 Related work

Various solutions have been developed to turn the computer's immediate feedback capabilities into a useful tool for the education of proof writing. Some offer total solutions, like Waterproof, whereas others seek to remedy specific issues.

Böhne and Kreitz, coauthors of [KFBK17], have developed a didactic method — opposed to a software solution — to explicitly guide the transition from Coq proofs to handwritten ones [BK18]. The gap between formal Coq proofs and informal textbook-style proofs is bridged by introducing three intermediate proof styles that gradually lower the level of formality. Students are first taught to write proofs in Coq — with custom, easier to learn tactics — and then to both translate between the different levels of formality as well as develop proofs from scratch at each level. The authors mention that a solution like Waterproof, which modifies Coq itself to make the proofs be more like handwritten proofs, would be very interesting, but that such a development would require a large amount of preparatory work. The teaching method, on the other hand, is cheap and flexible: it is easy to adapt to different domains and can be adjusted as a course is being taught.

Lurch [CM16] markets itself as a word processor that offers proof verification as an additional service, like a spellchecker; it is a total solution with its own deduction engine. The system does not force a specific syntax on the users, instead semantic information is obtained by having users manually annotate mathematical expressions as 'claims', 'reasons' or 'premises'. The mathematical expressions can be written using LaTeX, which is dynamically formatted. As of now, proof checking is limited to logic and set theory; proofs have to be written out in full detail, as by design Lurch has no automation facilities. Earlier versions included a computer algebra system for verification as well, but this has been removed due to insufficient precision.

The Diproche system [Car20, CLS22] is similar to Lurch: users can write out proofs in controlled natural language (German); the program parses the text and checks the proof using several proprietary deduction engines. Unique to Diproche is its ability to point out common mistakes and, in some cases, produce counterexamples. Currently, the Diproche system supports exercises in propositional logic, set theory, elementary number theory, axiomatic geometry and elementary group theory; the parsers and deduction systems had to be adjusted to each domain. Diproche runs on a remote web server, avoiding the problem of installation.

With the commercial web service Edukera [Edu12], users can advance a proof state simply by clicking buttons with inference rules. Because the interaction takes so little effort, there is a risk of students brute-forcing their way to a proof: instead of picking deliberate tactics, they click random buttons with the hope to advance the proof state, learning nothing. Having used Edukera in a course, Julien Narboux mentioned this risk at the Lean panel [BAN+21]; Patrick Massot, another participant, refrained from using Edukera in the past because even he himself started randomly pressing buttons out of frustration with the tool.

ProofWeb [MKvRW10] provides web access to a Coq proof assistant running on a central server and adds some additional features tailored to education. The user interface is similar to that of Coq itself, but the proof state can be displayed in different ways, like Gentzen's deduction trees or Fitch's flag-style proofs. The central server not only provides an easy way to access Coq, but also serves as a distribution point for exercises, meaning incomplete Coq files, and allows teachers to keep track of students' progress. A parser checks whether students are cheating by using Coq's powerful automation procedures instead of the intended tactics; like Böhne and Kreitz in [BK18], these tactics use a custom syntax that is easier to learn. ProofWeb use Coq version 8.2, which dates to 2009.

jsCoq [GAPJ17b] is an adaptation of Coq that runs purely in the browser using JavaScript; it can run offline without requiring any installation. jsCoq was not developed for teaching how to write proofs, but for reading mixed documents that combine human-readable text and verifiable Coq code; a nice example is a jsCoq version of the Software Foundations series [GAPJ17a] that allows users to directly see the effects of executing Coq code. Although it is possible to edit Coq files using jsCoq, there is no good way of storing the resulting files, so the system cannot be used for exercises that are to be handed in.

Waterproof was developed because none of the above solutions satisfied our specific needs. Lurch and Diproche come close since they allow users to write proofs in natural language. Both systems, however, use proprietary proof checkers, hence, they are more difficult to adapt for new courses and they miss out on the wealth of mathematics already formalized with proof assistants. Our decision to combine formatted text and Coq code in Waterproof files was partially inspired by jsCoq, but also by Jupyter Notebooks and Mathematica.

3 Choice of proof assistant

Waterproof is built on top Coq, but other proof assistants could have served as a suitable basis as well. Below we discuss three systems, Coq, Mizar and Lean, and motivate our design decision. For a broader overview, see the survey by Nawaz et al. which discusses 11 proof assistants [NML+19].

Coq is a stable, highly customizable proof assistant. Syntax, notations and tactics, all can be tailored to personal taste. One example is the Mathematical Components library [MT21] library, which was used to prove the four color theorem [Gon05] and odd order theorem [GAA+13]: it extends Coq's default syntax as well as its tactics in order to make the Coq files more similar to the original proofs. Custom editors can be created using SerAPI [GA16], a library for machine-to-machine interaction with Coq. Besides the Mathematical Components library, the Coq community maintains libraries covering algebra (Math Classes [SvdW11]), geometry (GeoCoq [BBB+18]), analysis, both constructive (C-CoRN [CFGW04]) and classical (Analysis compatible with Mathematical Components [ACM+18]), and homotopy type theory (HoTT Library [BGL+17], UniMath [VAG+]). The Coq project is open source and sees active development; recently the installation procedure has been simplified via the Coq Platform [CoqPltf].

The Mizar proof assistant was developed with the specific goal of being readable by ordinary mathematicians; it is guarded against a proliferation of different variants. Mizar's default syntax is closer to the mathematical vernacular than that of Coq or Lean, but cannot be customized to push the similarity further. Subsets are treated the same in Mizar as in standard mathematics, whereas Coq and Lean, because of their foundational systems, use classifying maps instead.

The mathematics formalized in Mizar is centrally organized in one big library, the Mizar Mathematical Library (MML) [MML], and covers topics like analysis, topology, measure theory, algebra, and graph theory. The development of custom libraries is discouraged. Although this policy prevents a proliferation of overlapping Mizar libraries, it restricts Mizar's use in education: it is desirable to slightly reformulate definitions and theorems to precisely match those in a textbook used by the course. The source code of the proof checker is not publicly available.

The Lean theorem prover is a relatively new system (founded in 2013) with a lot of momentum. Its central library `mathlib` [mC20] covers topics ranging from number theory to analysis. Part of Lean's promotional strategy has been to focus on formalizing 'fashionable' mathematics (see Buzzard's blogpost [Buz20]); the *Liquid Tensor Experiment* [Sch20] had a team of Lean experts formalize a technical, foundational theorem from Scholze' lecture notes on Analytic Geometry [SC20] which Scholze himself was uncertain about. The latest version Lean 4 [MU21] strengthens Lean's customizability: users can write their own parsers and complicated tactics within the Lean language itself; at the end of the Lean panel [BAN+21], Patrick Massot showed a custom parser he is writing in Lean 4 to make Lean's syntax more closely resemble a handwritten proof — a goal similar to that of coq-waterproof.

We decided to base Waterproof on the Coq proof assistant. Although Mizar proofs are closer to handwritten proofs by default, Coq and Lean can be sufficiently customized to surpass its similarity. Moreover, the source code of Mizar's proof engine is kept private, whereas Coq and Lean are both open source projects supported by large communities. Developing a custom parser might be easier with Lean 4 than with Coq, as the parser can be written in the Lean language itself, but, as a platform, Lean is less stable: Lean 4 will not be backwards compatible with Lean 3. The entirety of the `mathlib` library has to be ported to the new version and we are unable to guarantee the resources needed to keep Waterproof up-to-date in such an environment.

4 Design of coq-waterproof

Coq-waterproof enables students and mathematicians to write proofs in Coq in a style similar to handwritten proofs. This section elaborates on the customizations made to Coq in order to achieve this goal: from the extension of Coq's notations and tactics, to the addition of new reasoning capabilities.

It would be impossible to make these alterations without Coq's Ltac2 tactic language. The number of Ltac2 examples is limited, online and in literature, so we include ample of implementation details and code snippets.

4.1 Natural language tactics

First-year students often seem to think that mathematics is limited to formulas and logical expressions, they need to be made aware that language is a vital component of explaining one's reasoning and that it can be used with the level of rigour required in mathematics. We expect that their proofs consist of full, grammatically correct sentences, hence the tactics in coq-waterproof take this form as well. Compare the default Coq tactics

```
intro ε. intro ε_gt_0.
```

which introduce both a variable ε and the hypothesis that it is positive, with the equivalent tactics in coq-waterproof:

```
Take ε : ℝ. Assume that (ε > 0).
```

The meaning of coq-waterproof's tactics is immediately clear from their formulation.

The reformulation of Coq's default tactics into natural language ones required two decisions to be made: what notation pattern to use and what errors to return if a user's input does not match the current proof goal. The '`Take`'-tactic's notation pattern, for example, allows users to write multiple syntactically valid statements, such as:

```
Take ε : ℝ.
Take n : ℕ and x, y : ℝ.
```

To decide whether the current proof state actually requires all these user-specified variables to be introduced, the ‘`Take`’-tactic performs several checks. Note that more checks are required than with the equivalent default ‘`intro`’-tactic: ‘`Take ε : ℝ.`’ not only requires checking whether a variable ε needs to be introduced, like ‘`intro ε.`’, but also whether this variable should be a real number; such extensive control is possible because of the Ltac2 language.

4.1.1 Separate tactics for variables and propositions

Coq-waterproof uses separate tactics to introduce variables and add assumptions, like

```
Take ε : ℝ. Assume that (ε > 0).
```

whereas the default ‘`intro`’-tactic is used to introduce both variables and hypotheses, as in

```
intro ε. intro ε_gt_0.
```

The use of separate tactics better reflects mathematical vernacular. Note as well that using the default ‘`intro`’-tactic for hypotheses reduces a proof’s readability: in the above example, the hypothesis’s content, $\varepsilon > 0$, is obscured; the label ‘`ε_gt_0`’ hints at the content, but it could have been given any name by the user, like ‘`hypothesis1`’.

In order to have separate tactics for variables and propositions, one needs to be able to distinguish between the concepts in Coq: students should be prevented from writing e.g.

```
Assume that (ℝ).
```

This might seem trivial, but it is not: in Coq’s foundational system, namely type theory, elements of a set and labels of a proposition are manifestations of the same abstract concept. Fortunately, Coq’s standard mathematical library assigns sets and propositions to separate ‘universes’, so-called *sorts*, which can be queried. Such a convenient separation is not guaranteed: Coq’s Homotopy Type Theory library [BGL$^+$17], for example, places propositions in the same universe as singleton sets.

4.1.2 Ltac2 implementation of the ‘`Take`’-tactic

A large part of the ‘`Take`’-tactic’s code is shown in Listing 1; this subsection comments on some interesting aspects.

The notation pattern of the ‘`Take`’-tactic is specified using a single line, line 48; the syntactically valid tactic formulations specified are of the form:

```
Take x, y, ... : A and u, v, ... : B and ... .
```

The variables and sets put in by the user are assigned to the variable ‘`input`’ as a list, so in the above example ‘`input`’ equals ‘`[([x,y,...],A),([u,v,...],B),...]`’.

First, it is checked whether the ‘`Take`’-tactic can be applied at all (lines 35–39). The current goal is matched against a ∀-statement (lines 35–36); in case of a match, it is checked whether the goal is not to show an implication (line 39). In Coq, an implication is a special instances of universal quantification, namely one quantified over a proposition; precisely those are filtered out by line 39. If the goal is discovered to be an implication, lines 41–42 inform the user that the ‘`Assume`’-tactic should be used, as discussed in Section 4.1.1.

The majority of the remaining code serves to check whether the user-specified variables match the ∀-statements in the proof goal; the actual introduction of variables only occurs in line 7, by invocation of the default ‘`intro`’-tactic. The large number of checks also serves to provide the user with detailed feedback in case of a mistake. If, for example, the goal is to show that

$$\text{for all } n \in \mathbb{N}, x, y \in \mathbb{R}, \text{ if } 0 < x < y \text{ then } x^n < y^n ,$$

then the semantically incorrect tactics

```
Local Ltac2 intro_ident (id : ident) (type : constr) :=
    lazy_match! goal with
    | [ |- forall _ : ?u, _] =>
        let ct := Aux.get_coerced_type type in
        (* Check whether we need a variable of type [type], including coercions of [type]. *)
        match Aux.check_constr_equal u ct with
        | true  => intro $id
        | false => Control.zero (TakeError (too_many_of_type_message type))
        end
    | [ |- _] => Control.zero (TakeError (too_many_of_type_message type))
    end.

Local Ltac2 intro_per_type (pair : ident list * constr) :=
    match pair with
    | (ids, type) =>
        lazy_match! goal with
        | [ |- forall _ : ?u, _] =>
            (* Check whether [u] is not a proposition. *)
            let sort_u := Aux.get_value_of_hyp u in
            match Aux.check_constr_equal sort_u constr:(Prop) with
            | false =>
                (* Check whether we need variables of type [type], including coercions of [type]. *)
                let ct := Aux.get_coerced_type type in
                match Aux.check_constr_equal u ct with
                | true  => List.iter (fun id => intro_ident id type) ids
                | false => Control.zero (TakeError (expected_of_type_instead_of_message u type))
                end
            | true  => Control.zero (TakeError (of_string "Tried to introduce too many variables."))
            end
        | [ |- _ ] => Control.zero (TakeError (of_string "Tried to introduce too many variables."))
        end
    end.

Local Ltac2 take (x : (ident list * constr) list) :=
    lazy_match! goal with
    | [ |- forall _ : ?u, _] =>
        (* Check whether [u] is not a proposition. *)
        let sort_u := Aux.get_value_of_hyp u in
        match Aux.check_constr_equal sort_u constr:(Prop) with
        | false => List.iter intro_per_type x
        | true  => Control.zero (TakeError (of_string "‘Take ...’ cannot be used
                       to prove an implication (⇒). Use ‘Assume that ...’ instead."))
        end
    | [ |- _ ] => Control.zero (TakeError (of_string "‘Take ...’ can only be used
                      to prove a ‘for all’-statement (∀) or to construct a map (→)."))
    end.

Ltac2 Notation "Take" input(list1(seq(list1(ident, ","), ":", constr), "and")) :=
    panic_if_goal_wrapped ();
    take input.
```

Listing 1: Implementation of the ‘Take’-tactic.

```
Take n : ℕ and x, y, z : ℝ.
Take n : ℕ and x, y : ℝ and z : ℂ.
```

both result in an error, the respective error messages are

“Tried to introduce too many variables of type ℝ.” and “Tried to introduce too many variables.” .

4.2 Enforced signposting

Computers are great at bookkeeping, humans not so much: mathematicians need constant reminders on what is currently being shown, which case is being considered, etc. Handwritten proofs are full of such signposts to prevent readers from getting lost; proofs written in proof assistants contain hardly any, because they were not intended to be read by humans, but by a computer.

If a proof written with a proof assistant does contain signposts, they are usually comments. In the Coq proof in Figure 1, for example, comments are used to indicate which case is being considered:

```
destruct cases as [ε_lt_two | two_le_ε].
-(* Case ε < 2. *)
  ...
-(* Case ε ≥ 2. *)
  ...
```

Although comments like these keep a proof readable, they are optional, so students will often neglect to write them; neither do we wish to accidentally communicate to students that signposting one's proof is optional as well.

In coq-waterproof, signposting your proofs is mandatory: at certain points, the proof can only be advanced with a tactic that informs the reader about the proof's state. The result is a readable proof where signposts are placed on the same footing as the other proof elements:

```
Either (ε < 2) or (ε ≥ 2).
- Case (ε < 2).
  ...
- Case (ε ≥ 2).
  ...
```

The enforcement of signposting is achieved by inserting small subgoals that can be solved using specific tactics; the formulation of these tactics act as signposts. After a case distinction is performed using ‘`Either (ε < 2) or (ε ≥ 2).`’, the next subgoal (which is visible to the user) is *not*

```
there exists a : ℝ, a : [0,4) ∧ 4 - ε < a
```

as it would be in regular Coq, but

```
Add the following line to the proof:
  Case (ε < 2).
```

The proof can only be advanced by following this instruction and doing so reverts the goal back to the original existence statement.

4.2.1 Implementation

Behind the scenes, the ‘`Either ... or`’-tactic puts the subgoal of each branch in a ‘wrapper’ which can be removed by the correct ‘`Case`’-tactic. The implementations of the ‘wrapper’ and both tactics are shown in Listing 2.

The ‘wrapper’ combines information of the case being considered and the original goal G into a new type (line 2). It is inductively defined by a map ‘`wrap`’ which takes proofs of G and produces proofs of the wrapped goal (line 3); by induction, we also have a map the other way around, called ‘`unwrap`’ (lines 4–5). Because the wrapper is a *private* type (line 2), induction is only allowed to be performed inside the surrounding module (lines 1–6). Coq's notation system is used to render a wrapped goal as an instruction to use the corresponding 'Case'-tactic (lines 8–10); note the doubling of spaces in line 10.

The 'Either ... or'-tactic first attempts to find a proof that the cases specified by the user indeed cover all scenarios (lines 14–15); if successful, a case distinction is performed (line 19) and each subgoal is wrapped by applying the 'unwrap' map (lines 20–21), the corresponding case is given as the first argument. Application of the 'Case'-tactic with the correct input restores the original goal using the 'wrap' map (lines 31–33).

Other coq-waterproof tactics, like 'Take' or 'Assume', are prevented from manipulating wrapped goals using the 'panic_if_goal_wrapped' subroutine. If the goal is wrapped, it throws an error urging the user to follow the instruction given by the wrapper. The implementation is not shown in Listing 2, but it is invoked in line 27 and in Listing 1, line 49.

```
1   Module Case.
2       Private Inductive Wrapper (A G : Type) : Type :=
3           | wrap : G -> Wrapper A G.
4       Definition unwrap (A G : Type) : Wrapper A G -> G :=
5           fun x => match x with wrap _ _ y => y end.
6   End Case.
7
8   Notation "'Add' 'the' 'following' 'line' 'to' 'the' 'proof:' 'Case' ( A )." :=
9       (Case.Wrapper A _) (at level 99, only printing,
10          format "'[ ' Add  the  following  line  to  the  proof: ']' '//'   Case  ( A ).").
```

```
11  Ltac2 either_or (t1:constr) (t2:constr)
12      := let hint_databases := Some (load_databases global_decidability_database_selection) in
13      let h_id := Fresh.in_goal @h in
14      let attempt () := assert ({$t1} + {$t2}) as $h_id;
15                        ...
16      in
17      match Control.case attempt with
18      | Val _ => let h_val := Control.hyp h_id in
19                 destruct $h_val;
20                 Control.focus 1 1 (fun () => apply (Case.unwrap $t1));
21                 Control.focus 2 2 (fun () => apply (Case.unwrap $t2))
22      | Err exn => Control.zero (CaseError "Could not find a proof that
23                      the first or second statement holds.")
24      end.
25
26  Ltac2 Notation "Either" t1(constr) "or" t2(constr) :=
27      panic_if_goal_wrapped ();
28      either_or t1 t2.
```

```
29  Ltac2 case (t:constr) :=
30      lazy_match! goal with
31      | [|- Case.Wrapper ?v _] =>
32          match Aux.check_constr_equal v t with
33          | true => apply (Case.wrap $v)
34          | false => Control.zero (CaseError "Wrong case specified.")
35          end
36      | [|- _] => Control.zero (CaseError "No need to specify case.")
37      end.
38
39  Ltac2 Notation "Case" t(constr) := case t.
```

Listing 2: Implementation of the case-wrapper (top), 'Either ...or'-tactic (middle) and the 'Case'-tactic (bottom).

4.3 Mathematical notation

The notation used by coq-waterproof has been kept as close to mathematical convention as possible. Coq's standard library uses symbols which are similar to mathematical notation, but not quite the same:

the symbols '`/\`' and '`R`' are used to denote conjunction and the real numbers, whereas coq-waterproof uses '∧' and 'ℝ', see Figure 2. By default, function application in Coq is written as '`f x`', coq-waterproof allows users to write the familiar '`f(x)`'.

The notation used by Coq and coq-waterproof has to be expressed using Unicode, naturally limiting the extent to which the broad range of mathematical notation can be emulated. Notation which makes heavy use of sub- or superscript like

$$\lim_{n\to\infty} a_n = p \qquad \text{or} \qquad \sum_{k=0}^{\infty} \frac{1}{2^k} = 2$$

has to be expressed in some alternative way.

Both Coq and coq-waterproof use the notation '`x : ℝ`' to express that the variable x is a real number; it can be read as the mainstream set-theoretic notation '`x ∈ ℝ`', but, from a theoretical viewpoint, they are different. The :-notation occurs throughout Coq, so we did not dare to attempt to replace it by the ∈-symbol in coq-waterproof; it would be easier for students to learn that 'Waterproof just uses : instead of ∈' than for them to be confronted with two different symbols.

4.3.1 Subset membership

The set-theoretic symbol ∈ can also be used to denote membership of a subset, it is logically valid to write $x \in [0,4)$ if we already have that $x \in \mathbb{R}$, but this is not the case for the :-notation in Coq. If $x : \mathbb{R}$, it is theoretically invalid to write $x : [0,4)$, one has to use $0 \leq x < 4$ instead. This presents a problem if one indeed wants to tell students that the use of ':' in Waterproof and '∈' in ordinary mathematics is simply a difference in notation.

Coq-waterproof (ab)uses Coq's notation system to still allow statements like '`x : [0,4)`' to be written, see for example the existence statement in the proof goal in Figure 2:

```
there exists a : ℝ, a : [0,4) ∧ 4 - ε < a.
```

Because some Coq users might object to this abuse of the type-theoretical :-notation, it has been made optional and isolated from the rest of coq-waterproof.

Implementation

A '`subset`' in coq-waterproof, like '`[0,4)`', is defined as a record type with a single component '`pred`': its *classifying predicate* (see Listing 3, line 1). As it is customary to work directly with such predicates in Coq, subsets are automatically transformed into their predicates as needed; the coercion is included in the definition via the '`:>`'-symbol in line 1.

The notation '`x : [0,4)`' is defined (in line 3) as application of the '`pred`'-component of '`[0,4)`' to the term '`x`'; evaluating '`x : [0,4)`' results in the property '`0 ≤ x < 4`'. The '` : subset_scope`'-part at the end of line 3 places the notation in a newly declared scope (line 2); users can decide for themselves whether to open this scope or not.

Coq needs a little help using a statement like '`x : [0,4)`' as a hypothesis to prove '`2*x-1 < 7`'. The subroutine '`simpl_member_subset`' (lines 4–10) turns any hypothesis like '`x : [0,4)`' into the corresponding classifying property, in this case '`0 ≤ x < 4`'; it is prepended to the automation procedure that coq-waterproof uses to verify statements like '`2*x-1 < 7`'.

4.3.2 Expanding notations to their full definitions

Mathematical notation allows us to express complicated ideas succinctly, but to prove statements about them, one needs to be able to expand them into their full definitions. In order to show, for example, that '`4 is an upper bound of [0,4)`', one has to provide a proof for the underlying logical formula

'`for all x : ℝ, x : [0,4) ⇒ x ≤ 4`'.

In coq-waterproof, there are two ways to convert notations into their underlying definitions; the first is to use a coq-waterproof tactic that lets the user reformulate the goal manually. In the above example, starting the proof of the statement '`4 is an upper bound of [0,4)`' with

```
Record subset (X : Type) := as_subset { pred :> X -> Prop }.
```

```
Declare Scope subset_scope.
Notation "x : A" := ((pred _ A) x) (at level 70, no associativity) : subset_scope.
```

```
Ltac2 simpl_member_subset () :=
    repeat (
        match! goal with
        | [ h : (pred _ _) _ |- _ ] => simpl in $h
        | [ |- _ ] => ()
        end
    ).
```

Listing 3: Implementation of the subset type (above), membership notation (middle) and the subroutine which evaluates this notation in hypotheses (below).

```
We need to show that (for all x : ℝ, x : [0,4) ⇒ x ≤ 4).
```

transforms the goal into '`for all x : ℝ, x : [0,4) ⇒ x ≤ 4`'. This solution does require users to know the underlying definition, and they should know its formulation perfectly: any mistake and the above tactic will be rejected because the current goal and its rephrasing do not match.

The second approach is to let Coq perform the expansion for you: the tactic

```
Expand the definition of upper bound.
```

automatically converts the goal '`4 is an upper bound of [0,4)`' into the underlying definition. To keep the proof readable, the user is forced to repeat the conversion result; enforcement happens via the techniques discussed in Section 4.2. In this case, the user is forced to write:

```
That is, write the goal as (for all x : ℝ, x : [0,4) ⇒ x ≤ 4).
```

The second method is the preferred way to expand notations in Waterproof, as, currently, there are no ways for students to find the underlying definitions on their own. Students do need to be informed which part of the notation '`4 is an upper bound of [0,4)`' to put instead of the dots in

```
Expand the definition of ... .
```

For this purpose, we use underscores. For example, the above notation is always displayed as

'`4 is an _upper bound_ of [0,4)`',

corresponding to the use in

```
Expand the definition of upper bound.
```

The user can still use the notation without underscores in their proof.

Implementation

The second method requires that each notation introduced comes with its own tactics for expanding said notation. A lot of tactics have to be defined and all of them have to force the user to repeat the expanded definition in order to keep the proof readable.

Coq-waterproof provides a framework for the easy creation of such tactics; Listing 4 shows how the framework is used to define the tactics that can expand the '`4 is an upper bound of [0,4)`'-notation (lines 1–7). The notation for an upper bound is introduced in line 1; the term '`is_upper_bound`' has been defined previously and contains the underlying definition. Lines 2 and 3 define tactics which 'unfold' the '`is_upper_bound`' term: they replace its occurrence in the proof goal (line 2) or a hypothesis (line 3) by the underlying definition. Both tactics are to be fed into the '`expand_def_framework`' subroutine (lines

8–16); based on a third argument, the framework executes one of the two tactics and then applies the goal-wrapping technique from Section 4.2 to force the users to repeat the expanded definition. Lines 4–5 define the unfolding tactic intended for the user; the tactics in lines 2 and 3 are combined with the framework and a third argument ‘`cl`’. The tactic notation specifies that the ‘`cl`’-argument is optional, meaning that the tactic can be written in two ways:

```
Expand the definition of upper bound.
Expand the definition of upper bound in (i).
```

In the first case, we have ‘`cl = None`’ and ‘`cl = Some i`’ in the second; correspondingly, the framework will execute either lines 11–12 or 13–15. Lines 6–7 define an alternative formulation for the same tactic.

```
1  Notation "M 'is' 'an' 'upper' 'bound' 'of' A" := (is_upper_bound A M) (at level 69).
2  Local Ltac2 unfold_is_upper_bound    ()          := unfold is_upper_bound.
3  Local Ltac2 unfold_is_upper_bound_in (h : ident) := unfold is_upper_bound in $h.
4  Ltac2 Notation "Expand" "the" "definition" "of" "upper" "bound" cl(opt(seq("in", "(", ident, ")")))
5   := Waterproof.tactics.unfold.expand_def_framework unfold_is_upper_bound unfold_is_upper_bound_in cl.
6  Ltac2 Notation "Expand" "the" "definition" "of" "an" "upper" "bound" cl(opt(seq("in", "(", ident, ")")))
7   := Waterproof.tactics.unfold.expand_def_framework unfold_is_upper_bound unfold_is_upper_bound_in cl.
```

```
8  Ltac2 expand_def_framework (unfold_goal : unit -> unit) (unfold_hyp : ident -> unit) (cl : ident option)
9   := panic_if_goal_wrapped ();
10     match cl with
11     | None    => unfold_goal ();
12                  ap_goal_unwrap ()
13     | Some cl => let h_constr := Control.hyp cl in (* throws error if ident not found in hypotheses *)
14                  unfold_hyp cl;
15                  ap_hyp_unwrap h_constr
16     end.
```

Listing 4: Implementation of the ‘`is an upper bound`’-notation and the tactics used to expand its occurrences to the full definition (above), and the framework provided by coq-waterproof for the creation of such tactics (below).

4.4 Automated proof finding

The Analysis 1 course is meant to teach students how to write rigorous proofs, but this does not mean that every step has to be justified in full: the basic, often algebraic properties of the real numbers learned in high school do not require further explanation. Such basic claims, however, still need to be checked for mistakes, so the proof assistant has to try and find proofs for these statements on its own.

Automated proof finding is also used in the formalization of research level mathematics. Special tactics are created to automatically prove trivial, yet technical, statements, freeing mathematicians — both those writing and reading the proof — to focus on the interesting steps. The Lean library mathlib [mC20], for example, allows users to write ‘`by continuity`’ which will automatically attempt to prove that a function is continuous.

Note that such automation tactics still require active involvement from the user: they have to decide which tactic might be able to solve the proof goal.

Coq-waterproof uses automated proof finding *implicitly*: users only have to worry about the validity of their basic claims, not the justifications. Whereas the case distinction $\varepsilon < 2 \vee \varepsilon \geq 2$ in default Coq requires a justification in the form of the ‘`lra`’-tactic,

```
assert (ε < 2 \/ 2 <= ε) as cases by lra.
destruct cases as [ε_lt_two | two_le_ε].
```

coq-waterproof simply allows users to state

```
Either (ε < 2) or (ε ≥ 2).
```

Various other coq-waterproof tactics use implicit automation to check users' assertions, e.g.

```
It holds that (...).
We conclude that (...).
It suffices to show that (...).
```

If external lemmas or theorems are required for these assertions, they can be added with the prefix 'By (...)':

```
By (...) it holds that (...).
By (...) we conclude that (...).
By (...) it suffices to show that (...).
```

4.4.1 Customization using hint databases

Which statements should be considered 'trivial' varies throughout a course: at the start of Analysis 1, students explicitly have to show that $\sup[0,4) = 4$, but afterwards the same statement may be used without further explanation to show more complicated statements.

Default Coq allows users to control the strength of its automated proof finding tactic 'auto' with hint databases. These databases contain the lemmas and tactics which the automation tactic can use to solve the proof goal. The core database contains basic lemmas, specialized databases like bool and real contain techniques for solving problems with boolean logic and real numbers. The more hint databases are included, the stronger the automation tactic becomes.

Coq-waterproof also uses hint databases to customize its automation system, but the way databases are selected is a bit different. Normally, the hint databases are explicitly given as arguments to the 'auto'-tactic, like

```
auto with core bool real.
```

Because the automated proof finding in coq-waterproof is used implicitly, this approach is not possible. Instead, hint databases can be activated by including a separate line, like

```
Require Import Waterproof.load_database.RealNumbers.
```

From this point on, the implicit automation will use hints from a number of databases related to the reals; lines like these are usually placed at the start of a document where multiple files are imported.

4.4.2 Shielding goals from automation

To enable students to manipulate equations like they are used to from high school, coq-waterproof's automation system needs to be quite strong; on the other hand, the system should not be able to solve the exercises intended for students. As such, coq-waterproof shields certain goals from being solved automatically.

Implementation

We decided to shield universal and existential quantifications, conjunctions and disjunctions *unless* a statement is so easy that it can be solved with a restricted version of the automated proof finding algorithm; this is to filter out statements like $\forall n \in \mathbb{N}, n + 0 = n$ from being shielded. Lines 2–3 in Listing 5 define the attempt to solve the goal using the restricted algorithm; it is performed in line 23. If this attempt fails, a second, full attempt is performed (line 25). The second attempt (defined in lines 5–21) checks whether the goal is a $\forall$-,$\exists$-statement, conjunction or disjunction (lines 9–13) and, if this is the case, throws an error, shielding the goal from being solved.

```
Local Ltac2 actual_waterprove (prop: constr) (lemmas: (unit -> constr) list) (shield:bool) :=
    let first_attempt () := run_automation prop lemmas 3
        (Some ((@subsets)::(@classical_logic)::(@core)::[])) false
    in
    let second_attempt () :=
        match shield with
        | true => match global_shield_automation with
                  | true => (* Match goal with basic logical operators *)
                            lazy_match! goal with
                            | [ |- forall _, _ ] => fail_automation None
                            | [ |- exists _, _ ] => fail_automation None
                            | [ |- _ /\ _] => fail_automation None
                            | [ |- _ \/ _] => fail_automation None
                            | [ |- _] => ()
                            end
                  | false => ()
                  end
        | false => ()
        end;
        let databases := ... in
        run_automation prop lemmas global_search_depth databases global_enable_intuition
    in
    match Control.case first_attempt with
    | Val _ => ()
    | Err exn => match Control.case second_attempt with
                 | Val _ => ()
                 | Err exn => fail_automation (Some (Control.goal()))
                 end
    end.
```

Listing 5: Implementation of shielding goals from automated proof finding.

4.5 (In)equality chains

An important part of Analysis 1 is learning how to find upper bounds; proofs are often concluded by a chain of inequalities like $|f(x) - L| \leq \cdots < \varepsilon$. Some proof assistants support this style of notation (e.g. Lean's 'calculational' proofs), but Coq does not, so we had to add this feature to coq-waterproof.

The coq-waterproof example in Figure 2 shows inequality chains being used to prove the statements $0 \leq a$ and $4 - \varepsilon < 3$ via:

```
We conclude that (& 0 < 4 - 1 < 4 - ε/2 = a).
...
We conclude that (& 4 - ε ≤ 4 - 2 = 2 < 3).
```

4.5.1 Implementation

The top segment in Listing 6 shows how the notation for (in)equality chains was implemented in Coq. The starting '&'-symbol, see line 11, indicates that what follows should be interpreted as a chain; the chain itself starts with a term 'x' followed by a number of linkages like '< y' or '= z', the notation for these is specified above in lines 3–9. The '..' in line 11 denotes that the notation is defined recursively; the string 'lz .. lw' matches to any number of linkages greater than one. The definition follows in line 12: linkage 'ly' is first linked to the term 'x' via the 'chain_base' function, the other linkages are attached recursively via the function 'chain_link'. The chain in its entirety is fed into the 'total_statement' function, which transforms the chain, a data structure holding the user's input, into a meaningful mathematical statement, namely the proposition stating that every relation in the chain holds individually: the chain

```
(& 0 < 4 - 1 < 4 - ε/2 = a)
```

for example, is transformed into the proposition

$$(0 < 4 - 1) \ \wedge \ (4 - 1 < 4 - \varepsilon/2) \ \wedge \ (4 - \varepsilon/2 = a) \, .$$

The different kinds of linkages cannot be attached to each other freely: it is not mathematically valid to append a chain '$0 < x = y$' with '$> z$'. Coq-waterproof prevents users from writing such chains by only implementing the '`chain_link`' function for types of chains and linkages which are mathematically valid. The '`chain_link`' function is implemented as the property of a typeclass (line 14); whenever the proof checker encounters a '`chain_link`' function, it attempts to find an implementation with matching argument types. In lines 17–18, for example, a version of the '`chain_link`' function is specified for a chain containing only $=$-signs and a linkage of the $<$- or $\leq$-kind.

Typeclasses are also used to dynamically assign meaning to the $<$- and $>$-symbol used: if at some point an order is defined for some hitherto unordered set, it can quickly be made available for use in coq-waterproof's inequality chains. Lines 20–23 in Listing 6's bottom segment show the definition of the typeclass whose attributes provide ways to interpret the order symbols used in the chain notation. An instance implementing these functions for the natural numbers is shown in lines 25–28; these four lines are all what needs to be included for an order to be used by the inequality chains. Line 30 shows how to inform the proof assistant that the implementation of the typeclass is required in some function.

There are two disadvantages to using typeclasses: coercions have to be added manually and the error messages thrown are too difficult for students to understand. Because the proof assistant attempts to find an implementation with matching argument types, it has no way of knowing that a statement $n < 1/2$ with $n \in \mathbb{N}$ should be read as a comparison between real numbers, where the embedding $\mathbb{N} \hookrightarrow \mathbb{R}$ is used implicitly. As such, multiple implementations have to be added that use the embedding explicitly. If no matching interpretation can be found, for example when a students attempts to compare vectors in $\mathbb{R}^2$, a cryptic error is thrown. Ideally, we would catch this error and send a more readable one to the user, but we were unable to do so.

The implementation of the (in)equality chains could not be separated completely from the rest of the library. In the '`We conclude that`'-tactic, a piece of code is included to check whether a chain like `(& 0 < 4 - 1 < 4 - ε/2 = a)` actually matches what needs to be shown, in this case $(0 < a)$. Secondly, the automated proof finding system first splits a chain into its components before proving them separately; doing so improves the quality of the feedback: instead of receiving an error stating:

"Failed to show $(0 < 4 - 1) \wedge (4 - 1 < 4 - \varepsilon/2) \wedge (4 - \varepsilon/2 = a)$."

one receives the more detailed

"Failed to show $(4 - 1 < 4 - \varepsilon/2)$."

4.6 Manipulating negations

In a previous course, our students are taught De Morgan's laws as one of the basic ways to manipulate logical formula's; they should be able to convert statement

$$\neg\exists N \in \mathbb{N}, \forall k \in \mathbb{N}, k \geq N \Rightarrow a_k < L \quad \text{into} \quad \forall N \in \mathbb{N}, \exists k \in \mathbb{N}, (k \geq N) \wedge (a_k \geq L)\ . \tag{1}$$

Coq's automated proof finding system is only able to verify the initial conversion step, going from $\neg\exists N \in \mathbb{N}, P(N)$ to $\forall N \in \mathbb{N}, \neg P(N)$, after that, human guidance is needed. Coq-waterproof adds a new, recursive tactic which is able to verify an arbitrary number of nested invocations of De Morgan's laws; the conversion above is verified instantaneously.

4.6.1 Implementation

If the LHS of the conversion (1) is given as a hypothesis and the user states that the RHS then also holds, the tactic added by coq-waterproof needs to prove the implication

$$\Big(\neg\exists N \in \mathbb{N}, \forall k \in \mathbb{N}, k \geq N \Rightarrow a_k < L\Big) \quad \Longrightarrow \quad \Big(\forall N \in \mathbb{N}, \exists k \in \mathbb{N}, (k \geq N) \wedge (a_k \geq L)\Big)\ . \tag{2}$$

The implication is shown recursively, part of the tactic's implementation is shown in Listing 7.

```
Declare Scope chain_scope.
Delimit Scope chain_scope with chain.
Notation "< y"  := (chain_lt, y) (at level 69, y at next level) : chain_scope.
Notation "<= y" := (chain_le, y) (at level 69, y at next level) : chain_scope.
Notation "≤ y"  := (chain_le, y) (at level 69, y at next level) : chain_scope.
Notation "= y"  := (chain_eq, y) (at level 69, y at next level) : chain_scope.
Notation "> y"  := (chain_gt, y) (at level 69, y at next level) : chain_scope.
Notation ">= y" := (chain_ge, y) (at level 69, y at next level) : chain_scope.
Notation "≥ y"  := (chain_ge, y) (at level 69, y at next level) : chain_scope.
(* Full chain *)
Notation "& x ly lz .. lw" :=
  (total_statement (chain_link .. (chain_link (chain_base x ly%chain) lz%chain) .. lw%chain))
  (at level 70, x at next level, ly at next level, lw at next level).
```

```
Class ChainLink (A B C : Type) := chain_link : A -> B -> C.
 #[export] Instance link_ec_eq_inst   (T : Type) : ChainLink (EqualChain T) (EqualRel * T) (EqualChain T)
   := link_ec_eq T.
 #[export] Instance link_ec_less_inst (T : Type) : ChainLink (EqualChain T) (LessRel  * T) (LessChain T)
   := link_ec_less T.
 ...
```

```
Class OrderInterpretation (T : Type) :=
  { less_rel_to_pred : LessRel -> T -> T -> Prop
  ; grtr_rel_to_pred : GreaterRel -> T -> T -> Prop
  }.

#[export] Instance order_interpretation_nat : OrderInterpretation nat :=
  { less_rel_to_pred rel x y := match rel with | chain_lt => (x < y) | chain_le => (x <= y) end
  ; grtr_rel_to_pred rel x y := match rel with | chain_gt => (x > y) | chain_ge => (x >= y) end
  }.

Fixpoint lc_total_statement (T : Type) `{! OrderInterpretation T} (c : LessChain T) : Prop :=
  ...
```

Listing 6: Implementation of (in)equality chains: chain notation at the top, use of type classes to selectively implement the ‘chain_link’ function in the middle, and use of type classes to dynamically interpret the <- and >-symbols at the bottom.

First, the tactic inspects the shape of the implication and applies a lemma that reduces it to a simpler one. In the case of implication (2), it applies a lemma called '`not_ex_all_func`' (see lines 22–23) which states that

$$\Big(\forall x \in A, \neg P(x) \Rightarrow Q(x)\Big) \qquad \Longrightarrow \qquad \Big((\neg \exists x \in A, P(x)) \implies (\forall x \in A, Q(x)) \Big) .$$

for any set A and predicates P and Q over A . This lemma is shown earlier on in the file; similar statements for the other logical operators, negated and non-negated, are shown as well.

After the application of lemma '`not_ex_all_func`', it remains to show that

$$\forall N \in \mathbb{N}, \Big(\big(\neg \forall k \in \mathbb{N}, k \geq N \Rightarrow a_k < L\big) \quad \Longrightarrow \quad \big(\exists k \in \mathbb{N}, (k \geq N) \wedge (a_k \geq L)\big)\Big) .$$

A variable $N \in \mathbb{N}$ is introduced and the tactic is invoked again to show the simpler implication inside. This continues until in the end it remains to show that $\neg\ a_k < L \implies a_k \geq L$.

To prove these implications that require domain-specific knowledge, like $\neg a_k < L \implies a_k \geq L$, the automated proof finding system is called, but it is supplied only with hint databases that contain hints related to negations (lines 26–29). Which negation databases are included here depends on which hint databases are added to the automation system in general: the command

```
Require Import Waterproof.load_database.RealNumbers.
```

from Section 4.4.1 also adds a negation database with hints for (negated) order relations on the reals.

4.7 Suggestions for how to proceed

When students get stuck with a proof, coq-waterproof can give them some basic hints on how to proceed. The suggestions are limited to the 'mechanical' aspect of proof writing: for example, if the goal is to show a $\forall x \in \mathbb{R}, \ldots$-statement, executing the command

```
Help.
```

will return a message suggesting user to introduce a variable with the '`Take (...)`'-tactic:

```
The goal is to show a 'for all'-statement (∀).
Introduce an arbitrary variable of type ℝ.
Use 'Take ... : ...'.
```

Coq-waterproof is unable to provide hints for proof steps that require some level of ingenuity, like how to show an inequality, or which variable to pick to prove an existence statement.

5 Design of the Waterproof editor

The second component of the Waterproof software is a custom editor designed specifically with education in mind. The editor is often simply refered to as 'Waterproof', below we discuss the features that distinguish the Waterproof editor from other Coq editors.

5.1 Mixed documents

As mentioned in the introduction, Waterproof files combine formatted text and Coq code in a single document. The inclusion of human-readable texts makes such mixed documents ideal for communicating mathematical results, think of Jupyter Notebooks or Mathematica. To emphasize the mixed nature of their contents, we also refer to Waterproof files as *notebooks*.

The formalized mathematical statements and proofs expressed in the Coq language are organized into *code blocks*. A Coq instance running in the background verifies their contents; the *SerAPI* library is used to facilitate communicate between the editor and the Coq instance.

Similarly, the human-oriented explanations, examples, or comments are organized into text blocks, the text can be styled with Markdown and allows for LaTeX-formatted expressions to improve readability. Waterproof allows for the creation of a special kind of text block called *hint blocks*: they reveal a hidden text when clicked and a bold outline makes them stand out from regular text blocks.

```
Local Ltac2 solve_by_manipulating_negation_in (h_id : ident) :=
    let h := Control.hyp h_id in
    (* Check whether h is a proposition. *)
    let type_h := Aux.get_value_of_hyp h in
    let sort_h := Aux.get_value_of_hyp type_h in
    match Aux.check_constr_equal sort_h constr:(Prop) with
    | false => Control.zero (NegationError "Can only manipulate negation in propositions.")
    | true =>
        let attempt () :=
            revert $h_id;
            solve [ repeat ( first
                [ (* finish proof *)
                  exact id
                | lazy_match! goal with
                  (* without negation *)
                  ...
                  | [ |- (forall x, @?p x) -> (forall x, @?q x)] =>
                      apply (all_func _ $p $q); let x_id := Fresh.in_goal @x in intro $x_id
                  ...
                  (* with negation *)
                  ...
                  | [ |- not (exists x, @?p x) -> (forall x, @?q x)] =>
                      apply (not_ex_all_func _ $p $q); let x_id := Fresh.in_goal @x in intro $x_id
                  ...
                  end
                | (* Try context specific manipulation, e.g. negating order relations *)
                  let g := Control.goal () in
                  let hint_databases := Some (load_databases global_negation_database_selection) in
                    run_automation g [] 1 hint_databases false
                ] ) ]
        in
        match Control.case attempt with
        | Val _ => ()
        | Err exn => Control.zero (NegationError "Failed to solve by manipulating negation.")
        end
    end.

Ltac2 solve_by_manipulating_negation () :=
    match! goal with
    | [ h : _ |- _ ] => solve_by_manipulating_negation_in h
    end.

Global Hint Extern 1 => ltac2:(solve_by_manipulating_negation ()) : classical_logic.
```

Listing 7: Implementation of negation manipulation tactic. Top: attempts to show a statement using a single hypothesis h_id, middle: try the upper tactic with all hypotheses, bottom: addition of the middle tactic to the classical-logic hint database.

5.1.1 Exercise sheets

For use in a course setting, where students have to complete the proofs in a prepared Waterproof document as part of their homework, it is important that parts of the notebook cannot be altered by students: they might strengthen the automated proof finding system or accidentally delete half of the assignment. The latter sounds like a stretch, but happens all the time in a different course where we use Jupyter Notebooks as exercise files.

Waterproof notebooks can be converted into so-called exercise sheets which can only be altered in designated *input sections*. Teachers can insert these sections into a notebook during the editing process, they are indicated by big, square, horizontal brackets (shown in Figure 4). In the non-converted notebook version, teachers can use these sections to try out their own solutions without risk: all content in input sections is deleted upon conversion.

5.2 Layout — limited proof progress window

The layout of the Waterproof editor is still very similar to that of the CoqIDE, except for one detail: the proof progress window by default no longer shows the assumptions and variables introduced. Users are forced to pay more attention to the proof text already written, and with coq-waterproof's natural language tactics, the text will include all the relevant information. The removal also serves our goal to make writing proofs in Waterproof similar to writing proofs by hand, with pen and paper you also has to keep track of hypotheses yourself. In the settings menu one can re-enable the displaying hypotheses, but one can also opt not to show the current goal either, pushing users even more to signpost their proofs by writing

```
We need to show that (...).
```

5.3 Convenience

Waterproof offers several features to aid users with proof writing: a drop down menu for common mathematical symbols and a panel with all coq-waterproof tactics for reference.

The drop down menu for (mathematical) Unicode symbols pops up when typing a string starting with '\'. For example, see Figure 4, typing '`\in`' offers a collection of symbols `∈, ∞, ℤ` and ∩ whose encoded reference names all start with '`\in`' (namely `\in`, `\infty`, `\integers` and `\intersection`). The encoded names match the convention used in LaTeX whenever possible. This way of typing mathematical Unicode symbols is similar to the CoqIDE's solution: typing '`\in`' followed by pressing the shortcut '`Shift + Space`' replaces the string '`\in`' by the symbol '∈'. The drop down menu is easier to use for beginners however, as it does not require them to know the LaTeX-abbreviations by heart.

The reference panel for coq-waterproof tactics is also shown in Figure 4. It can be made to (dis)appear by clicking the hammer icon in the top-right corner. The tactics are accompanied by a short explanation and next to each tactic two symbols are shown. Clicking the top one copies a tactic to the clipboard, the bottom one inserts the tactic directly at the cursor in a code block that is being edited.

5.4 Styling

The user interface for Waterproof is based on *Electron*. This way, Waterproof is an extension of the chromium web browser, and it is largely written in JavaScript and in the Vue.js framework. A big advantage is that styling of the user interface can be done for a large part with CSS and thus be altered with minimal effort.

We have decided to give Waterproof a minimalist appearance, its default color palette is dominated by blues and white. Thanks to the power of CSS, however, we can also offer a 'dark mode', dominated by black, for the students' late-night proving needs. In the future we wish to provide more customization features via the 'settings' menu.

5.5 Compatibility with existing Coq ecosystem

The Coq proof assistant has an active community and its members have invented all kinds of supplementary programs to improve the Coq experience. We wish to *add* to this ecosystem and thus made it possible to convert Waterproof notebooks to and from regular Coq files.

Waterproof's styled text blocks are converted into *coqdoc* comments, Coq's own solution for the inclusion of stylized text in source files. Coqdoc comments, however, cannot replicate Waterproof's text styling in full: to render the coqdoc comments in their stylized format, the files containing them can be compiled into either a LaTeX or a HTML file, but bold typefaces cannot be used in the LaTeX document and LaTeX expressions cannot be used in the HTML document.

Because regular Coq files can be converted into Waterproof notebooks, any Coq editor can be used to create them. The Waterproof editor is then only necessary to improve the formatting.

6 Student experience

As stated in the introduction, Waterproof has been tested for the last couple of years in the Analysis 1 course at the TU/e. For several homework assignments, student groups could hand in a completed Waterproof exercise sheet instead of the usual pen-and-paper proofs. There was a small benefit to using Waterproof: if a proof was judged to be correct by Waterproof, full points would be awarded, even though an instructor might have done so for a similar proof on paper. Waterproof is not yet perfect, it might verify a logical step that an instructor thinks to be too large, but the students should not be blamed for trusting the machine's response. Use of Waterproof was optional and no time was reserved specifically for teaching students how to work with Waterproof; a tutorial exercise sheet was available and students could ask questions about Waterproof during guided homework sessions, although only a few instructors had experience with Waterproof themselves. At the start of the 2021/22 course, 16 student groups (≈ 64 students) handed in homework written in Waterproof; 9 groups still used it for the final assignment.

Courses at the TU/e are concluded with a student survey which allowed us to ask some questions regarding students' experience with Waterproof. First we asked students how much they liked working with Waterproof on a scale from 1 ('not at all') to 5 ('very much'); the results were varied, the 21 responses received showed an almost uniform distribution. This matches our personal experience: some students quickly abandoned Waterproof, confused or discouraged by its syntax or the required strictness, others were so enthusiastic that they asked if they could hand in all their homework using Waterproof.

Next we asked what suggestions students had for improving Waterproof. Multiple students replied that they struggled to learn how to use Waterproof whilst also having to study the mathematical theory of the course. One student wrote:

> *"I had a look at Waterproof but I was busy learning the theory*
> *so I didn't also have the capacity of struggling with the syntax."*

Note that the above comments refer to a previous iteration of coq-waterproof's syntax, not the version presented in this article; whether students will still experience such troubles with the improved syntax will have to show in the 2022/23 course. Nevertheless, we will also create instructional videos explaining how to use Waterproof, as suggested by another student.

Students also reported struggling with Waterproof's automation system: some assertions had to be reformulated in equivalent ways before Waterproof could verify them. One student stated that:

> *"The program often acts probabilistic as to whether or not it will accept a step in a proof, e.g.:*
> $a+1=1+a$ *would be accepted, yet* $a+2-1=2-1+a$ *would not be accepted."*

Waterproof can actually verify that $a+2-1=2-1+a$ holds, but we do recognize the issue: Waterproof can show that $\sqrt{a}^2 = a$ for $a \geq 0$, but not that $\sqrt{a}^2 + 0 = a$. Although Waterproof is deterministic, the system seeming to behave irrational is indeed very confusing to students and undermines their trust in a program that we claim can check the correctness of their proofs. Getting Waterproof's automation system more in line with students' and teachers' expectations will be a top priority in its future development.

7 Discussion

When grading the homework exercises, we started to notice that students were using Waterproof's tactic formulations in their handwritten proofs as well; the explicit phrasing of these sentences helped to clarify the logical structure of their arguments. Since Waterproof only accepts logically valid proofs, the student proofs written in Waterproof were, of course, logically clear and part of this clarity seems to transfer to pen-and-paper proofs.

Currently, we have evaluated Waterproof based on small student surveys, one-on-one conversations with students, and tutors' personal experience. In a future study, the benefits of Waterproof should be investigated more systematically. In particular, we would like to investigate the effects of Waterproof with a greater number of students, with a clear control group, both looking at grades and by performing an in-depth analysis of their handwritten proofs.

It would be good to compare Waterproof to other teaching methods that incorporate proof assistants, like the pedagogical method described in [BK18]: its solution requires only minimal alterations of Coq's default syntax, whereas to be most useful for different courses, the coq-waterproof library needs to incorporate the mathematical notation of those mathematical fields.

Waterproof should, however, not solely be judged on its educational effectiveness: coq-waterproof also enables ordinary mathematicians to write formal proofs in a style familiar to them. We hope that this aids in the adoption of proof assistants among mathematicians.

7.1 Future improvements

In its current state, the coq-waterproof library and Waterproof editor are already useful tools for utilizing the power of proof assistants in the classroom, yet there remain aspects to be improved.

7.1.1 Automation system

The main focus of future developments will be to improve coq-waterproof's automation systems. Currently, the system is simultaneously both too strong and too weak: some exercises that we want students to prove themselves, can be shown automatically, whereas some 'basic' computations cannot. Like the example in the previous section: Waterproof can show that $\sqrt{a}^2 = a$ for $a \geq 0$, but not that $\sqrt{a}^2 + 0 = a$. The system also seems to be unable to utilize hypotheses that are bound up in $\wedge$-statements: given that $A \wedge B$ holds, Waterproof is unable to show directly that $A \wedge (3 < 5)$.

We would also like the future automation system to be able to tell whether a certain lemma or hypothesis is necessary in finding a proof. Currently the system will mark the following statement as correct:

```
By (IVT) it holds that (1 + 1 = 2).
```

Although the statement is true from a purely logical perspective (the implication $\texttt{IVT} \implies 1 + 1 = 2$ holds vacuously by truth of the conclusion), the natural language expression suggests that the Intermediate Value Theorem is necessary to show that $1 + 1 = 2$. This is of course false, so the statement should be rejected.

7.1.2 Additional parsing options

In our attempt to imitate natural language proofs within Coq, we ran into the limitations of its notation system. We have tried to work-around these, but the proper solution would be to extend the default parser using a Coq-plugin.

One example of such a work-around can be found in the tactic for obtaining a concrete variable from an existence statement. For example, given some $\varepsilon > 0$, one wants to use the fact that $\exists N \in \mathbb{N}, \forall n \geq N, |1/n - 0| < \varepsilon$ (labeled `(i)`) as follows:

```
Obtain N according to (i), so for N : ℕ it holds that
  (for all n : ℕ, n ≥ N ⇒ |1/n - 0| < ε).
```

The second occurrence of N is redundant, but we need it to make the notation work. Preferably we would write

```
Obtain N : ℕ according to (i) such that (for all n : ℕ, n ≥ N ⇒ |1/n - 0| < ε).
```

but Coq does not let us because the proposition at the end is ill-formed: when the above sentence is checked, the variable N has not yet been introduced, so the proposition

```
(for all n : ℕ, n ≥ N ⇒ |1/n - 0| < ε)
```

does not make sense to Coq. The upper notation works because the second N is actually a dummy variable, and the subphrase

```
for N : ℕ it holds that (for all n : ℕ, n ≥ N ⇒ |1/n - 0| < ε)
```

secretly stands for the predicate P given by

$$P(N) := (\forall n : \mathbb{N}, n \geq N \implies |1/n| < \varepsilon) \ .$$

Although this trick works for our purposes, one could easily undermine its intended meaning by picking a different dummy variable.

7.1.3 Waterproof editor

Waterproof's current design is still mainly based on that of regular theorem provers, but might not be optimal for teaching students how to write mathematical proofs. Different layouts and features could be tried to make writing proofs in Waterproof more closely resemble writing proofs on paper. One way in which we did change Waterproof's layout compared to other theorem provers is by hiding hypotheses from the proof progress window (by default).

Similarly, we are still looking for a design solution to the expanding of definitions and notations. Waterproof offers two ways to expand a definition (see Section 4.3.2): the first requires one to already precisely know the expanded formulation, the second leaves a trace in the final proof text that would not be present in a pen-and-paper proof. The first option would be fine, *provided* that there was a convenient way to find the expanded formulation on your own. A possible solution would be to show the expanded formulation when the user hovers over them with the cursor.

The maintenance of a custom editor takes up a considerable amount of resources; preferably, the Waterproof editor would be replaced by a minimal plugin that extends existing Coq editors. What currently sets the Waterproof editor apart from others, is the ability to combine formatted comments with mathematical expressions and Coq code in a single document; this is very useful e.g. for creating exercise sheets and theory files, and can also benefit the readability of existing Coq library files. When mixed documents will be supported by other editors in the future, Waterproof's development can focus on adding just those elements required for education: being able to 'lock' certain parts of a notebook, enabling students to find the expanded formulation of a definition, hiding the separate panel with hypotheses.

7.2 Advanced feedback

We have put a lot of effort into providing hints and feedback to students, but it still mostly focuses on the logical soundness of their proofs. As pointed out in the introduction (issue P3), students need different kinds of feedback as well. Again, think of students that neglect to use lemmas and try to derive everything from first principles, they require feedback that considers the proof as a whole and goes beyond checking line-by-line correctness.

There are no clear-cut solutions to these problems, but we expect that some AI systems will be involved in solving them in the future. *Tactician* [ZBP$^+$21], for example, is a machine learning plugin for Coq that can suggest the next couple of steps to a user; it could be used to generate hints for students, but a good hint goes beyond simply revealing the answer step-by-step: it nudges a student in the right direction whilst still requiring the student to think for themselves.

8 Conclusion

Proof assistants are being used more and more to teach students how to write mathematical proofs, but these programs were not designed for education. Novel users have a hard time learning the syntax and the proof writing skills obtained with theorem provers do not automatically transfer to pen-and-paper proofs. *Waterproof* is an extension of the Coq proof assistant that seeks to address these issues.

One part of Waterproof is a custom library called *coq-waterproof* which allows users to write Coq proofs in a style that closely matches handwritten proofs. Proof steps can be written in natural language, as opposed to Coq's cryptic keywords (so-called *tactics*), and common mathematical notation can be used. At crucial steps, coq-waterproof forces users to signpost their proofs to keep them readable. Coq-waterproof also allows users to write chains of (in)equalities, a feature missing from default Coq. The library is written with the Ltac2 tactic language and the exposition in this article includes many code examples showcasing the language.

The second part of Waterproof is a custom editor designed with education in mind. Waterproof documents not only contain formal Coq proofs, but combine these with formatted text including LaTeX expressions. The human-readable text can give further explanation about a mathematical topic or be used to describe e.g. a homework assignment, an incomplete Coq proof which is left as an exercise to the reader. To prevent students from accidentally deleting exercises, a Waterproof document can be 'locked', meaning that it can only be altered in designated areas. The editor allows for the easy typing of mathematical symbols with a drop-down menu; an optional side panel offers a quick overview of coq-waterproof tactics. Waterproof also removes some features that provide too much support for students, like a separate panel that neatly tracks all hypotheses and variables in the current proof; students should learn to gather this information from the proof text themselves, as they would with a handwritten proof.

The Waterproof software has been successfully used to supplement teaching in the Analysis 1 course at the TU/e. Preliminary observations suggest that using Waterproof assists students in clarifying the logical structure of their proofs, including those produced with pen and paper. A future study should investigate these claims more thoroughly by systematically analyzing student grades and the quality of their proofs.

We hope that the ideas presented in this article may add to the development of educational software that helps students learn how to write mathematical proofs, as well as make proof assistants more accessible to ordinary mathematicians.

References

[ACM+18] Reynald Affeldt, Cyril Cohen, Assia Mahboubi, Damien Rouhling, and Pierre-Yves Strub. Classical Analysis with Coq. In *The Coq Workshop 2018*, 2018.

[BAN+21] Jasmin Blanchette, Jeremy Avigad, Julien Narboux, Heather Macbeth, Gihan Marasingha, and Patrick Massot. Panel: Teaching with proof assistants. Lean Together 2021, Jan 2021.

[BBB+18] Michael Beeson, Pierre Boutry, Gabriel Braun, Charly Gries, and Julien Narboux. GeoCoq, June 2018.

[BBG+15] Grzegorz Bancerek, Czesław Byliński, Adam Grabowski, Artur Korniłowicz, Roman Matuszewski, Adam Naumowicz, Karol Pąk, and Josef Urban. Mizar: State-of-the-art and Beyond. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics*, pages 261–279, Cham, 2015. Springer International Publishing.

[BGL+17] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, page 164–172, New York, NY, USA, 2017. Association for Computing Machinery.

[BK18] Sebastian Böhne and Christoph Kreitz. Learning how to Prove: From the Coq Proof Assistant to Textbook Style. In Pedro Quaresma and Walther Neuper, editors, Proceedings 6th

International Workshop on *Theorem proving components for Educational software,* Gothenburg, Sweden, 6 Aug 2017, volume 267 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–18. Open Publishing Association, 2018.

[Buz20] Kevin Buzzard. Where is the fashionable mathematics? https://xenaproject.wordpress.com/2020/02/09/where-is-the-fashionable-mathematics/, 2020. [Online; accessed 29-June-2022].

[Car20] Merlin Carl. Number Theory and Axiomatic Geometry in the Diproche System. In Pedro Quaresma, Walther Neuper, and João Marcos, editors, Proceedings 9th International Workshop on *Theorem Proving Components for Educational Software,* Paris, France, 29th June 2020, volume 328 of *Electronic Proceedings in Theoretical Computer Science*, pages 56–78. Open Publishing Association, 2020.

[CFGW04] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the Constructive Coq Repository at Nijmegen. In Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, editors, *Mathematical Knowledge Management*, pages 88–103, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[CLS22] Merlin Carl, Hinrich Lorenzen, and Michael Schmitz. Natural Language Proof Checking in Introduction to Proof Classes - First Experiences with Diproche. In João Marcos, Walther Neuper, and Pedro Quaresma, editors, Proceedings 10th International Workshop on *Theorem Proving Components for Educational Software,* (Remote) Carnegie Mellon University, Pittsburgh, PA, United States, 11 July 2021, volume 354 of *Electronic Proceedings in Theoretical Computer Science*, pages 59–70. Open Publishing Association, 2022.

[CM16] Nathan Carter and Kenneth Monks. From Formal to Expository: Using the Proof-Checking Word Processor Lurch to Teach Proof Writing. In Rachel Schwell, Aliza Steurer, and Jennifer F. Vasques, editors, *Beyond Lecture: Resources and Pedagogical Techniques for Enhancing the Teaching of Proof-Writing Across the Curriculum*, pages 299–309. Mathematical Association of America, Washington, DC 20090-1112, 2016.

[Coq22] The Coq Development Team. The Coq Proof Assistant, January 2022.

[CoqPltf] The Coq Platform Development Team. Coq Platform.

[dMKA$^+$15] 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.

[Edu12] Edukera. http://edukera.com/, 2012.

[GA16] Emilio Jesús Gallego Arias. SerAPI: Machine-Friendly, Data-Centric Serialization for Coq. Technical report, MINES ParisTech, October 2016.

[GAA$^+$13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 163–179, Rennes, France, July 2013. Springer.

[GAPJ17a] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq-powered version of Software Foundations. https://jscoq.github.io/ext/sf/, 2017. [Online; accessed 29-June-2022].

[GAPJ17b] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq: Towards hybrid theorem proving interfaces. In Serge Autexier and Pedro Quaresma, editors, *Proceedings of the 12th*

Workshop on User Interfaces for Theorem Provers, Coimbra, Portugal, 2nd July 2016, volume 239 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–27. Open Publishing Association, 2017.

[Gon05] Georges Gonthier. A Computer-Checked Proof of the Four Colour Theorem. Jan 2005.

[HH11] Martin Henz and Aquinas Hobor. Teaching Experience: Logic and Formal Methods with Coq. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 199–215, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[KFBK17] Maria Knobelsdorf, Christiane Frede, Sebastian Böhne, and Christoph Kreitz. Theorem Provers as a Learning Tool in Theory of Computation. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, page 83–92, New York, NY, USA, 2017. Association for Computing Machinery.

[mC20] 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.

[MKvRW10] Hendriks Maxim, Cezary Kaliszyk, Femke van Raamsdonk, and Freek Wiedijk. Teaching logic using a state-of-art proof assistant. *Acta Didactica Napocensia*, 3(2):35–48, June 2010.

[MML] The Mizar Mathematical Library. `http://mizar.org/`.

[MT21] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, Jan 2021.

[MU21] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[Nip12] Tobias Nipkow. Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 24–38, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[NML$^+$19] M. Saqib Nawaz, Moin Malik, Yi Li, Meng Sun, and Muhammad Ikram Ullah Lali. A Survey on Theorem Provers in Formal Methods. *CoRR*, abs/1912.03028, 2019.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL*, volume 2283 of *LNCS*. Springer Berlin, Heidelberg, 2002.

[RZ05] Krzysztof Retel and Anna Zalewska. Mizar as a Tool for Teaching Mathematics. *Mechanized Mathematics and Its Applications*, 4, Special Issue on 30 Years of Mizar:35–42, March 2005.

[SC20] Peter Scholze and Dustin Clausen. Lectures on Analytic Geometry. `http://www.math.uni-bonn.de/people/scholze/Analytic.pdf`, 2020.

[Sch20] Peter Scholze. Liquid tensor experiment. `https://xenaproject.wordpress.com/2020/12/05/liquid-tensor-experiment/`, 2020. [Online; accessed 29-June-2022].

[SvdW11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011.

[TI21] Athina Thoma and Paola Iannone. Learning about Proof with the Theorem Prover LEAN: the Abundant Numbers Task. *International Journal of Research in Undergraduate Mathematics Education*, 8:64–93, July 2021.

[VAG$^+$] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. Unimath. available at `https://unimath.org`.

[ZBP$^+$21] Liao Zhang, Lasse Blaauwbroek, Bartosz Piotrowski, Prokop Černý, Cezary Kaliszyk, and Josef Urban. Online Machine Learning Techniques for Coq: A Comparison. In Fairouz Kamareddine and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 67–83, Cham, 2021. Springer International Publishing.