BACHELOR SOFTWARE ENGINEERING PROJECT

Software User Manual

Team Waterfowl

July 1, 2021

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

2IPE0 Software Engineering Project
QUARTILE IV - 2020-2021

Software User Manual

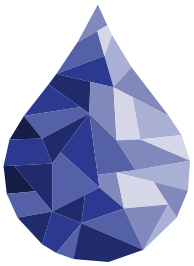
Team Waterfowl

Authors:
Adrien Castella *(1280880)*
Adrian Cucoş *(1327860)*
Cosmin Manea *(1298542)*
Noah van der Meer *(1116703)*
Lulof Pirée *(1363638)*
Mihail Ţifrea *(1317415)*
Tristan Trouwen *(1322591)*
Tudor Voicu *(1339532)*
Adrian Vrămuleț *(1284487)*
Yuqing Zeng *(1284835)*

Supervisor:
Gerard Zwaan, univ. lecturer
Customer:
Jim Portegies, asst. prof.

Version 0.3

July 1, 2021



Abstract

This manual explains the practical usage of the three deliverables of the *Waterfowl* project. It is divided into three sub-manuals, each tailored to the target audience of the respective deliverable. The first part is an extensive reference manual for all the tactics of the Waterproof tactics library. The second part explains the API of the Abstract Syntax Tree of Waterproof. The last part explains how Waterproof's graphical dependency installer and updater can be used.

Contents

1 General Introduction **9**
- 1.1 Intended readership . . . 9
- 1.2 Applicability . . . 9
- 1.3 How to use this document . . . 9
- 1.4 Related documents . . . 9
- 1.5 Problem reporting . . . 10

I Tactics Library **11**

2 Introduction **11**
- 2.1 Intended Readership . . . 11
- 2.2 Purpose . . . 11
- 2.3 How to use this part of the manual . . . 11
- 2.4 Conventions . . . 12
  - 2.4.1 Types . . . 12
  - 2.4.2 Examples . . . 13

3 Overview **13**
- 3.1 Tutorials . . . 13
- 3.2 Tactics Reference . . . 14
- 3.3 Appendices . . . 14

4 Tutorial **15**
- 4.1 Configuration of automation . . . 15
  - 4.1.1 Functional description . . . 15
  - 4.1.2 Cautions and warning . . . 15
  - 4.1.3 Procedures . . . 15
  - 4.1.4 Possible errors . . . 17
- 4.2 Installation through opam . . . 17
  - 4.2.1 Functional description . . . 17
  - 4.2.2 Cautions and warnings . . . 17
  - 4.2.3 Procedures . . . 17
  - 4.2.4 Possible errors . . . 18

5 Reference **19**
- 5.1 Apply . . . 19
  - 5.1.1 Functional Description . . . 19
  - 5.1.2 Formal Description . . . 19
  - 5.1.3 Examples . . . 19
  - 5.1.4 Cautions and Warnings . . . 19
  - 5.1.5 Possible Errors . . . 19
  - 5.1.6 Related Operations . . . 20
- 5.2 Assume . . . 21
  - 5.2.1 Functional Description . . . 21
  - 5.2.2 Formal Description . . . 21
  - 5.2.3 Examples . . . 21
  - 5.2.4 Cautions and Warnings . . . 23
  - 5.2.5 Possible Errors . . . 23

5.2.6 Related Operations . . . 23
5.3 Because . . . 24
5.3.1 Functional Description . . . 24
5.3.2 Formal Description . . . 24
5.3.3 Examples . . . 24
5.3.4 Cautions and Warnings . . . 25
5.3.5 Possible Errors . . . 25
5.3.6 Related Operations . . . 25
5.4 Choose such that . . . 26
5.4.1 Functional Description . . . 26
5.4.2 Formal Description . . . 26
5.4.3 Examples . . . 26
5.4.4 Cautions and Warnings . . . 27
5.4.5 Possible Errors . . . 27
5.4.6 Related Operations . . . 27
5.5 Choose . . . 28
5.5.1 Functional Description . . . 28
5.5.2 Formal Description . . . 28
5.5.3 Examples . . . 28
5.5.4 Cautions and Warnings . . . 29
5.5.5 Possible Errors . . . 29
5.5.6 Related Operations . . . 29
5.6 Contradiction tactics . . . 30
5.6.1 Functional description . . . 30
5.6.2 Formal description . . . 30
5.6.3 Examples . . . 30
5.6.4 Cautions and warnings . . . 30
5.6.5 Possible errors . . . 30
5.6.6 Related operations . . . 30
5.7 Define . . . 31
5.7.1 Functional Description . . . 31
5.7.2 Formal Description . . . 31
5.7.3 Examples . . . 31
5.7.4 Cautions and Warnings . . . 31
5.7.5 Possible Errors . . . 31
5.7.6 Related Operations . . . 31
5.8 Either . . . 32
5.8.1 Functional Description . . . 32
5.8.2 Formal Description . . . 32
5.8.3 Examples . . . 32
5.8.4 Cautions and Warnings . . . 32
5.8.5 Possible Errors . . . 32
5.8.6 Related Operations . . . 32
5.9 Help . . . 33
5.9.1 Functional Description . . . 33
5.9.2 Formal Description . . . 33
5.9.3 Examples . . . 33
5.9.4 Cautions and Warnings . . . 34
5.9.5 Possible Errors . . . 34
5.9.6 Related Operations . . . 34

5.10 It holds that . . . 35
5.10.1 Functional Description . . . 35
5.10.2 Formal Description . . . 35
5.10.3 Examples . . . 35
5.10.4 Cautions and Warnings . . . 36
5.10.5 Possible Errors . . . 36
5.10.6 Related Operations . . . 36
5.11 Rewrite equality . . . 37
5.11.1 Functional Description . . . 37
5.11.2 Formal Description . . . 37
5.11.3 Examples . . . 38
5.11.4 Cautions and Warnings . . . 38
5.11.5 Possible Errors . . . 39
5.11.6 Related Operations . . . 39
5.12 Rewrite inequality . . . 40
5.12.1 Functional Description . . . 40
5.12.2 Formal Description . . . 40
5.12.3 Examples . . . 40
5.12.4 Cautions and Warnings . . . 41
5.12.5 Possible Errors . . . 42
5.12.6 Related Operations . . . 42
5.13 Sets tactics . . . 43
5.13.1 Functional description . . . 43
5.13.2 Formal description . . . 43
5.13.3 Examples . . . 43
5.13.4 Cautions and warnings . . . 43
5.13.5 Possible errors . . . 44
5.13.6 Related operations . . . 44
5.14 Simplify what we need to show . . . 45
5.14.1 Functional Description . . . 45
5.14.2 Formal Description . . . 45
5.14.3 Examples . . . 45
5.14.4 Cautions and Warnings . . . 45
5.14.5 Possible Errors . . . 45
5.14.6 Related Operations . . . 45
5.15 Take . . . 46
5.15.1 Functional Description . . . 46
5.15.2 Formal Description . . . 46
5.15.3 Examples . . . 46
5.15.4 Cautions and Warnings . . . 46
5.15.5 Possible Errors . . . 47
5.15.6 Related Operations . . . 47
5.16 This follows by assumption . . . 48
5.16.1 Functional Description . . . 48
5.16.2 Formal Description . . . 48
5.16.3 Examples . . . 48
5.16.4 Cautions and Warnings . . . 48
5.16.5 Possible Errors . . . 49
5.16.6 Related Operations . . . 49
5.17 This follows by reflexivity . . . 50

5.17.1 Functional Description . . . 50
5.17.2 Formal Description . . . 50
5.17.3 Examples . . . 50
5.17.4 Cautions and Warnings . . . 50
5.17.5 Possible Errors . . . 50
5.17.6 Related Operations . . . 50
5.18 Unfold . . . 51
5.18.1 Functional Description . . . 51
5.18.2 Formal Description . . . 51
5.18.3 Examples . . . 51
5.18.4 Cautions and Warnings . . . 52
5.18.5 Possible Errors . . . 52
5.18.6 Related Operations . . . 52
5.19 We claim that . . . 53
5.19.1 Functional Description . . . 53
5.19.2 Formal Description . . . 53
5.19.3 Examples . . . 53
5.19.4 Cautions and Warnings . . . 53
5.19.5 Possible Errors . . . 53
5.19.6 Related Operations . . . 54
5.20 We conclude that . . . 55
5.20.1 Functional Description . . . 55
5.20.2 Formal Description . . . 55
5.20.3 Examples . . . 56
5.20.4 Cautions and Warnings . . . 56
5.20.5 Possible Errors . . . 56
5.20.6 Related Operations . . . 57
5.21 We know . . . 58
5.21.1 Functional Description . . . 58
5.21.2 Formal Description . . . 58
5.21.3 Examples . . . 58
5.21.4 Cautions and Warnings . . . 58
5.21.5 Possible Errors . . . 58
5.21.6 Related Operations . . . 59
5.22 We need to show . . . 60
5.22.1 Functional Description . . . 60
5.22.2 Formal Description . . . 60
5.22.3 Examples . . . 60
5.22.4 Cautions and Warnings . . . 60
5.22.5 Possible Errors . . . 61
5.22.6 Related Operations . . . 61
5.23 We prove by induction on . . . 62
5.23.1 Functional Description . . . 62
5.23.2 Formal Description . . . 62
5.23.3 Examples . . . 62
5.23.4 Cautions and Warnings . . . 62
5.23.5 Possible Errors . . . 62
5.23.6 Related Operations . . . 62
5.24 We show/prove both directions . . . 63
5.24.1 Functional Description . . . 63

5.24.2 Formal Description . . . 63
5.24.3 Examples . . . 63
5.24.4 Cautions and Warnings . . . 63
5.24.5 Possible Errors . . . 63
5.24.6 Related Operations . . . 64
5.25 We show both statements . . . 65
5.25.1 Functional Description . . . 65
5.25.2 Formal Description . . . 65
5.25.3 Examples . . . 65
5.25.4 Cautions and Warnings . . . 66
5.25.5 Possible Errors . . . 66
5.25.6 Related Operations . . . 66
5.26 Write as . . . 67
5.26.1 Functional Description . . . 67
5.26.2 Formal Description . . . 67
5.26.3 Examples . . . 67
5.26.4 Cautions and Warnings . . . 67
5.26.5 Possible Errors . . . 67
5.26.6 Related Operations . . . 68

II Waterproof Installer **69**

6 Introduction **69**
6.1 Intended readership . . . 69
6.2 How to use this part of the manual . . . 69
6.3 Purpose . . . 69

7 Overview **69**

8 Tutorials **69**
8.1 Install Waterproof dependencies . . . 69
8.1.1 Functional description . . . 69
8.1.2 Cautions and warnings . . . 69
8.1.3 Procedure: . . . 69
8.1.4 Possible errors . . . 70
8.2 Uninstall Waterproof dependencies . . . 72
8.2.1 Functional description . . . 72
8.2.2 Cautions and warnings . . . 72
8.2.3 Procedures . . . 72
8.2.4 Possible errors . . . 72
8.3 Update Waterproof . . . 72
8.3.1 Functional description . . . 72
8.3.2 Cautions and warnings . . . 72
8.3.3 Procedures . . . 72
8.3.4 Possible errors . . . 73

III Abstract Syntax Tree **74**

9 Introduction **74**

9.1 Intended readership . . . 74
9.2 How to use this part of the manual . . . 74
9.3 Purpose . . . 74

10 Overview **74**

11 Tutorials **74**
11.1 Syntax highlighting . . . 74
11.1.1 Functional description . . . 74
11.1.2 Preconditions . . . 74
11.1.3 Procedures . . . 74
11.1.4 Likely errors . . . 75
11.2 Extend AST-parser . . . 75
11.2.1 Functional description . . . 75
11.2.2 Preconditions . . . 75
11.2.3 Procedures . . . 75
11.2.4 Likely errors . . . 76
11.2.5 Cautions and warnings . . . 76
11.3 Pretty-printer . . . 76
11.3.1 Functional description . . . 76
11.3.2 Preconditions . . . 77
11.3.3 Procedures . . . 77
11.3.4 Likely errors . . . 77
11.4 Visitor pattern . . . 77
11.4.1 Functional description . . . 77
11.4.2 Preconditions . . . 77
11.4.3 Procedures . . . 77
11.4.4 Likely errors . . . 78
11.4.5 Cautions and warnings . . . 78

References **79**

A Error messages and recovery procedures **80**
A.1 Error messages for the tactics library . . . 80
A.2 Error messages for the installer . . . 83
A.3 Error messages for the AST . . . 83

B Glossary **85**

C Index **86**
C.1 List of all tactics library tutorials . . . 86
C.2 List of all tactics . . . 86
C.3 List of installer and updater tools . . . 86
C.4 List of Abstract Syntax Tree tools . . . 87

Document Status Sheet

Title	Software User Manual
Authors	Adrien Castella, Adrian Cucoş, Cosmin Manea, Noah van der Meer, Lulof Pirée, Mihail Ţifrea, Tristan Trouwen, Tudor Voicu, Adrian Vrămuleț, Yuqing Zeng
Version	0.3
Status	final
Creation Date	May 17, 2021
Last Modified	July 1, 2021

Document History log

June 29^{th}, 2021: Initial draft release (version 0.1).

July 30^{st}, 2021: First revision based on feedback received from the project managers (version 0.2).

July 1^{st}, 2021: Final revision (version 0.3).

1 General Introduction

Due to the nature of this project, this section provides a general introduction to the entire Software User Manual.

1.1 Intended readership

This manual explains the practical usage of each of the three deliverables of the *Waterfowl* SEP project. The manual is split into three separate parts since each of the respective deliverables is independent of the others and targets a different audience. Although more details about the intended readership are given for each part, a summary is provided here for reference.

The first part is an exhaustive reference manual for the Waterproof tactics library. The target audience of this part is comprised of the Coq users who use the library to write Coq proof scripts with a pen-and-paper-like notation. This part will explain the semantics and syntax of each tactic in the library. It will also provide example usages of each tactic in a simple context. The manual also lists common errors that tactics may raise and explains under which conditions they may occur.

The second part of the manual provides a walk-through for the graphical dependencies installer of Waterproof. This part is intended for general Windows users who need to install Waterproof. In addition, it is also designed for users updating Waterproof.

The last part explains the JavaScript API of the Abstract Syntax Tree parser of the Waterproof GUI. This part is aimed at developers who develop and maintain the JavaScript codebase of the Waterproof GUI.

1.2 Applicability

This manual applies to the latest version of the software delivered by the Waterfowl team on July 1st 2021. Specifically, for the tactics library, this applies to the `coq-waterproof` library[1], version 1.0.0, for the Coq Proving Assistant. This library can be found on GitHub at [13]. The library is also published to the Coq Package Index at [5] (which is also accessible through GitHub via the opam-coq-archive, see [12]).

1.3 How to use this document

This manual explains the practical usage of each of the three deliverables of the *Waterfowl* SEP project. As previously mentioned, this manual is split into three separate parts since each of the three deliverables is independent of the others and targets a different audience. Refer to the intended readership paragraph above to determine which part is of interest. Each part will have more precise instructions on how to use the manual, in addition to providing info on its intended readership.

1.4 Related documents

The User Requirements Document of the Waterfowl project [4] is strongly related to this manual. In the respective document, the requirements of the Waterfowl project are described and have been given a priority, indicating the necessity and importance of the requirement being implemented. The requirements specify what the user can do, whereas

[1] Also known as the *Waterproof tactics library*.

this manual helps the user to perform the possible actions and utilise the features specified in the User Requirements document.

The Software Design Document [2] of the Waterfowl project is also relevant. It contains a description of the design of the tactics library, alongside a detailed description of the overall functionality of all features that will be described throughout the rest of this manual.

Furthermore, the Software Transfer Document [3] has importance for the developer-focused sections. In this document, guides for setting up the developer environment are provided. These will not be repeated here. Instead, more details about this are given in the intended readership sections of each part.

1.5 Problem reporting

Bugs and problems found can be reported to the GitHub Issues page of the respective project. Issues for the tactics part (cf. Part I) can be reported at

github.com/impermeable/coq-waterproof/issues.

Issues for the dependencies installer (cf. Part II) can be reported at

github.com/impermeable/waterproof-dependencies-installer/issues.

Issues for the Abstract Syntax Tree (cf. Part III), as well as issues for the updater (cf. Part II) can be reported at

github.com/impermeable/waterproof/issues.

Alternatively, one can mail the maintainer Jim Portegies at

j.w.portegies@tue.nl

Part I

Tactics Library

2 Introduction

2.1 Intended Readership

This part of the user manual describes the syntax and semantics of the tactics in the Waterproof tactics library. The Waterproof tactics library is a standalone Coq library and can be used by any interested Coq user.

This part of the manual does not assume that the reader is familiar with the Waterproof GUI (for which the library was originally developed). However, the reader should be familiar with the basics of the Coq Proving Assistant ([8]). The reader should be familiar with concepts such as the "*proof state*", a "*goal*" and a "*tactic*". Also basic built-in tactics (such as `simpl`, `ring`, `assumption` and `reflexivity`) will be used in examples. For a basic introduction in Coq, see (for example) the first chapters of the book *Coq'Art* [1] or the Coq manual [14].

2.2 Purpose

The library adds a set of alternative tactics and notations to Coq. These tactics are designed to create proofs that resemble pen-and-paper Mathematics. Hence, the tactics are more similar to natural language than the built-in tactics of Coq. Proofs written using the Waterproof tactics library are reasonably interpretable to a mathematician who is unfamiliar with Coq.

Besides the custom tactics, the library also offers automation features that can automatically solve "trivial" steps in proofs. In a formal system such as Coq, it is often unavoidable to use more statements than one would in natural language, especially for obvious details. The tactics library reduces the need fir such trivial statements.

The tactics library is intended to be used within the already existing Waterproof software. However, it can also be used in any other Coq environment as a stand-alone Coq library.

This reference manual describes the syntax and semantics of each tactic. The explanations are pragmatic, and are given both at a formal and at an intuitive level. This manual explains the syntax of each tactic formally, but also provides examples to illustrate how the tactics can be used in practice. The manual also describes the configurability of the hints databases needed for some of the present tactics, together with the configurability of the search depth and of the `intuition` tactic. For more technical information regarding these concepts, please refer to the Software Design Document of the Waterfowl project [2].

2.3 How to use this part of the manual

This manual provides (1) a tutorial for configuring the automation, (2) a tutorial for installing the library via the opam tool, and (3) an extensive reference manual for all the tactics.

The tutorial for configuring the automation is provided in Section 4. This is a step-by-step walk-through that demonstrates the Coq commands needed to configure the automation.

The installation tutorial is provided in Section 4.2. This tutorial lists a series of command-line instructions that can be followed to install the library through the tool known as opam [11].

This tutorial requires a command line interface with the program opam installed, which is particularly suitable for the Linux and macOSX operating systems.

The reference manual for the tactics is ordered alphabetically. This part of the manual is especially relevant for users who need to find how to use a specific tactic, or who need to find which tactic can be used for a specific purpose. To locate a specific tactic, one can consult the table of contents () or the index C.2.

2.4 Conventions

The formal syntax of tactics is explained using conventions inspired by the Coq Reference Manual [15]:

- The `monospace font` is used to denote literal syntax keywords. Note that a final `.` is usually also part of the syntax.
- The $cursive\ font$ is used to denote where expression of a certain type should be written. These occurrences act as placeholders. For example, $ident$ signifies that an identifier should be entered at this position in the syntax.
- The notation $[content]^{rep}_{sep}$ is used to quantify the amount of repetitions possible for $content$. The brackets are not a literal part of the syntax, but only used to describe the syntax. This notation is used for lists in which individual items are described by $content$, or for an optional item $content$.

  In case lists are used, sep is replaced by a string that acts as a delimiter.

  rep is replaced by a token that describes the number of repetitions. The following tokens are used:

  - $?$: for optional terms. That is, zero or one items. This token is not used for lists.
  - $+$: for a list of one or more items.
  - $*$: for a list of 0 or more items.
- $[A|B|C|\dots]$ is used to denote that different alternatives are available. In the given example, the available alternatives are A, B, C and $\dots$.

2.4.1 Types

The following syntactic types are used:

- $ident$: an 'identifier': a name that can be used to refer to a hypothesis or a free variable. Also, a new name for a to-be introduced variable can be given as an ident. Idents may consist of alphabetical characters, an underscore ('_'), and digits. Note that the first characters may not be a digit. Also see [6].
- $constr$: a 'constructor'. This can be any mathematical expression, such as '`(x + 3 = 2 * y)`' (such expression are also known as $terms$ in Coq, see [7]), including the name of, for instance, a previously proven lemma. Constructors consisting of multiple symbols must be enclosed in parentheses.

  A reference to an existing hypothesis can also be used as a constructor. From a user's perspective, this usage is indistinguishable from using an identifier.

2.4.2 Examples

$$\texttt{MyTactic}\ [ident]^{+}_{,}\ .$$

Indicates the literal keyword `MyTactic`, followed by a list of identifiers. The identifiers are separated by `,` symbols. A final `.` indicates the end of the tactic. The following would be a valid invocation:

```
MyTactic some_ident, another_ident, some_other_ident.
```

$$\texttt{OtherTactic}\ int\ [\texttt{+}\ int]^{?}\ .$$

Indicates literal keyword `OtherTactic`, followed by an integer. Optionally, the keyword `+` and another integer can follow the first integer. As usual, a final '`.`' closes the tactic. Valid invocations are:

```
OtherTactic 13.
```

and

```
OtherTactic 13 + 6.
```

3 Overview

The tactics library part of the manual is subdivided into two main sections: (1) two tutorials and (2) a reference manual for the tactics.

3.1 Tutorials

The upcoming section (Section 4) describes two tutorials for the following tasks:

1. **Configuring the automation** (Section 4.1): this tutorial demonstrates how the behaviour of the automated tactics can be configured at runtime through Coq commands. Since only a limited set of databases is loaded by default, the amount of proofs that the automated tactics can solve is quite limited. This section is therefore relevant for readers who are working on a proof in a specific subdomain of mathematics. For example, the tactics will not be able to solve set-theoretic proofs without the database for sets.
2. **Installing the library through opam** (Section 4.2): this tutorial explains how the Waterproof tactics library can be installed from the opam-coq-archive [12] (which is the repository that describes the libraries found in the Coq Package Index [5]). It uses the command-line tool opam [11], which is used for most optional Coq libraries. This tutorial is relevant for Coq users who want to use and install the Waterproof tactics library as a stand-alone Coq library, without installing the other dependencies Waterproof dependencies [2]. Note that this tutorial uses the command line. This is convenient for Unix-based operating systems such as Linux and macOS, but may be more difficult in Windows.

[2] Windows users that desire to install Waterproof should probably consider using the Windows dependency installer for Waterproof, as described in Part II

3.2 Tactics Reference

The next section (Section 5) is the reference manual for the tactics. This section explains every tactic in the library individually, in alphabetical order. It lists tactics that have synonymous notations by their most common name. Each entry in the reference describes the following aspects of a specific tactic:

1. An intuitive description of the purpose of the tactic.
2. A formal description. This includes a precise, general description of the syntax. The conventions for the syntax explanations are given in Section 2.4.
3. One or more examples of the tactic used in a simple (partial) proof. The aim of these examples is to show how the syntax is used in practice, and how the tactic behaves within a concrete proof script.
4. A few recommendations that explain how to avoid common errors.
5. A list of possible warnings that the tactic could raise, and their causes.

3.3 Appendices

The end of this document provides three appendices:

- **Error messages and recovery procedures**: this appendix includes a list of all errors that can be raised by the tactics (see Section A.1).
- **Glossary**: the glossary provides a list of definitions of technical terms used in this manual, including terms related to the tactics library (see Section B).
- **Index**: the index provides tables with all tutorials regarding the tactics library (see Section C.1) and all tactics (see Section C.2). The latter table provides for each tactic (1) a very brief description of its purpose, and (2) a reference to the respective section. This index is especially useful if one knows what they want to achieve, but not the name of the tactic that can be used for their purpose.

4 Tutorial

4.1 Configuration of automation

This tutorial shows how the automated tactics can be configured.

4.1.1 Functional description

All automated tactics rely on three settings:

- **Databases**: this setting concerns the set of hint databases that the automation currently uses. These databases contain important theorems and lemmas for specific subdomains of mathematics. Without loading the required database, the automated tactics will not be able to apply those theorems when searching for a proof. More databases can be loaded and added to this set at runtime. Note that these are not databases in the usual sense, but a collection of lemmas that have already been proven[3].
- **Search depth**: this setting governs how many sequences of actions the automation can try out. For example, if the search depth is two, the automation will only attempt to finish the proof by using steps with a total cost of 2 or less. Naturally, a higher search depth makes the automation able to prove more goals, but also increases the running time of automated tactics.
- **Intuition**: this setting is a Boolean flag: it can be enabled or disabled. If it is enabled, then the automation will also use Coq's `intuition` algorithm in addition to the default automation algorithms used[4]). This setting is disabled by default, but enabling it can increase the set of goals that automated tactics can solve.

4.1.2 Cautions and warning

The tactics library must be installed on the local machine if one wants to use the mentioned automation features. Setting the search depth too high and/or loading too many databases can significantly increase the running time and memory needs of the automated tactics. This is especially noticeable when one makes a mistake, and the automation will attempt to prove a goal that is logically false. This can potentially make a computer run out of memory (i.e. the user interface of the computer 'freezes').

The only caution regarding enabling 'intuition' is that enabling it might prove too powerful for the intended usage. It may take non-trivial steps in order to prove a goal, which could lead to an obscured proof script. This is of course only relevant if good readability of the proof script is desired.

4.1.3 Procedures

Settings can be configured by importing a specific module of the `Waterproof` library. These setting-modules do not add more tactics or lemmas into the current environment: they only change the settings of the automation.

One can add more databases by importing a sub-module from `Waterproof.load_database`.

[3] For more information about the technical details of databases, please refer to the Software Design Document ([2])

[4] For more information about `intuition`, please refer to [2, Section 3.3.7]

For instance, one can add the database for real-number multiplication (see Table 1 for a complete list of hints databases) as follows:

```
Require Import Waterproof.load_database.Multiplication.
```

Database	Description
AbsoluteValue	Contains properties of the distance (i.e. absolute value function) defined on the real line.
Additional	Handles propositional logic.
All	Contains all the other databases in this table.
Exponential	Contains properties of the exponential and logarithm functions defined on the real line.
Multiplication	Contains properties of the multiplication operation defined on the real line.
Other	Currently empty. Can be easily populated with lemmas that do not fit any of the categories described in this table.
PlusMinus	Contains properties of the plus and minus operation defined on the real line.
reals	Not to be confused with the existing Coq library with the same name, but starting with a capital letter. This one is custom made and contains properties of the real numbers that do not fit the previous categories. Moreover, these properties are not found in the existing Coq library with the same name.
RealsAndIntegers	Contains the existing Coq library for dealing with reals (the one with the same name as the previous database) and the existing Coq libraries for dealing with integers. Note that if a claim can be proven for real numbers, the same claim cannot be proven for the integers when using the same lemmas.
Sets	Handles logic with sets.
SquareRoot	Contains properties for squaring real numbers.
ZeroOne	Handles operations which entail addition or multiplication by the real numbers zero or one.

Table 1: Table containing the name of each hints database, alongside a high-level description.

In order to set the search depth, import the corresponding file from

```
\waterproof \set_search_depth.
```

For example, setting it to 5:

```
Require Import Waterproof.set_search_depth.To_5.
```

Intuition can be enabled by importing the file "Enabled" from

```
\waterproof \set_intuition.
```

```
Require Import Waterproof.set_intuition.Enabled.
```

Similarly, importing the file "Disabled" from

```
\waterproof \set_intuition
```

disables intuition:

```
Require Import Waterproof.set_intuition.Disabled.
```

4.1.4 Possible errors

There are no documented possible errors for this tutorial.

4.2 Installation through opam

This tutorial describes how the Waterproof tactics library can be installed as a stand-alone tactics library.

Users of the Waterproof GUI [9] should note that the Windows 10 Waterfowl installer (cf. Part II of this document) already installs the tactic library.

4.2.1 Functional description

This tutorial described how the Waterproof tactics library can be installed from the official Coq Package Index [5] via the command-line. Note that the Coq Package Index describes the files from the opam-coq-archive [12]. These packages can be installed through the tool called opam [11]. This tutorial will describe the series of commands needed to make opam download and install the library.

4.2.2 Cautions and warnings

This tutorial uses the command-line, and assumes that both Coq and opam are already installed. This is convenient for users of the Bash shell in Unix-based operating systems such as Linux and macOS. However, for Windows users, it may require prior work to set up a command-line prompt with opam installed.

4.2.3 Procedures

It is assumed that the opam software (cf.[11]) and the Coq proof assistant are already installed on this machine.

The steps for installing the tactics library are the following:

1. Open a Bash terminal.
2. Run the command

   ```
   opam repo add coq-released https://coq.inria.fr/opam/released
   ```

   and wait for it to finish.
3. Run the command

   ```
   opam install -v coq-waterproof
   ```

   and wait for it to finish.

The last command from above installs and compiles the tactics library completely.

4.2.4 Possible errors

There are no applicable possible errors for this tutorial.

5 Reference

This section will explain how the individual tactics that are part of the Waterproof tactics library are to be used. The syntax, behaviour, and intent of each tactic will be described. Note that all tactics are defined within the Coq system.

5.1 Apply

5.1.1 Functional Description

The `Apply` tactic is used to apply an already proven result (either a hypothesis or a previously proven lemma), in order to prove the current goal. Formally, the tactic tries to match the current goal with the conclusion of the applied result, and then rewriting the goal using that particular result.

5.1.2 Formal Description

The syntax of the `Apply` tactic is

$$\texttt{Apply } constr,$$

where $constr$ is the statement that will be applied (it can be the name of a hypothesis, the name of a previously proven statement etc.).

5.1.3 Examples

Assume that there exists a proven lemma called `my_refl`, which states that

$$\forall n \in \mathbb{N}, n = n.$$

Consider now the following example:

```
Goal forall n : nat, n = n.
Proof.
    Take n : nat.
    Apply my_refl.
```

Here, the `Take` tactic was used to introduce the variable $n \in \mathbb{N}$ (cf. 5.15). Then, after calling the tactic

```
Apply my_refl.
```

the goal is automatically proven, as the goal can be matched with the conclusion of the lemma `my_refl`.

5.1.4 Cautions and Warnings

When applying a particular statement is not possible, the `Apply` tactic fails with an error.

5.1.5 Possible Errors

When the `Apply` tactic is called with a parameter $constr$, and it is not possible to match the current goal with the conclusion of $constr$,t hen Coq raises the error

$$\texttt{Unable to unify } statement\_of\_constr \texttt{ with } goal.$$

where $goal$ is the statement of the current goal and $statement\_of\_constr$ is the actual statement in $constr$.

5.1.6 Related Operations

This tactic is not particularly related to any of the other tactics in the new tactics library.

5.2 Assume

5.2.1 Functional Description

The `Assume` tactic can be used to start the proof of an implication ($\Rightarrow$). It introduces the premise (the left hand side expression) as an implication. For example, to prove $A \Rightarrow B$, one assumes that the premise A holds, and shows that B follows. In Waterproof, this assumption can be written as follows:

```
Assume h : A.
```

Note that `h` is the name assigned to the hypothesis. The only function of this name is to refer to the hypothesis as a variable, and one is free to choose any unoccupied name.

In case the premise is a conjunction of multiple propositions itself, each proposition can be assumed independently. But the conjunctions can also be preserved, one is free to choose either option. For example, if the goal is $(A \wedge B) \wedge (C \wedge D) \Rightarrow E$, then one can assume two hypotheses $(A \wedge B)$ and $(C \wedge D)$. But `Assume` also allows to introduce A, B, C and D. Also combinations are possible, e.g. assuming A, B and $(C \wedge D)$. To specify how such a hypothesis should be introduced, one explicitly gives each hypothesis a name and adds its type (its proposition), e.g.:

```
Assume h_a : A and h_b: B and h_cd: (C /\ D).
```

5.2.2 Formal Description

The formal definition of the syntax is:

$$\texttt{Assume}\ [ident\ :\ constr]^{+}_{\texttt{and}}.$$

Where the $ident$s are the names of the hypotheses, and the $constr$ the types (i.e. actual propositions) of the hypotheses. $constr$ should be Gallina code that evaluates to an expression of Coq-type $Prop$.

An alternative syntax exists, which simply replaced the `Assume` keyword with the `such that` keyword:

$$\texttt{such that}\ [ident\ :\ constr]^{+}_{\texttt{and}}.$$

5.2.3 Examples

Basic example Consider the following proof goal:

```
Goal forall n, n = 1 /\ n = 2 -> False.
```

After applying `Take n:nat.`, one will be left with an implication. `Assume` can now be used to introduce the premise:

```
Assume n_is_one : (n = 1) and n_is_two : (n = 2).
```

After executing the above line, the proof state will be:

```
n: nat
n_is_one: n = 1
n_is_two: n = 2

(1/1)
False
```

That is, the remaining goal is to derive the conclusion of the hypothesis (in this case the proposition `False`), and the available hypotheses are `n`, `n_is_one` and `n_is_two`

Mixed Conjunction Breakdown Consider the following goal:

```
Goal (A /\ B) /\ (C /\ D) -> E.
```

for some propositions `A`, `B`, `C`, `D` and `E`. There are five possible ways to introduce the hypotheses:

- One hypothesis: `(A /\ B) /\ (C /\ D)`.
- Two hypotheses: `(A /\ B)` and `(C /\ D)`.
- Three hypotheses: `A`, `B` and `(C /\ D)`
  or `(A /\ B)`, `C` and `D`.
- Four hypotheses: `A`, `B`, `C` and `D`.

To use one option with three hypotheses, one can write:

```
Assume h_a:A and h_b:B and h_cd:(C /\ D).
```

Which results in the following proof state:

```
A, B, C, D, E: Prop
h_a: A
h_b: B
h_cd: C /\ D

(1/1)
E
```

Such That Notation The last example demonstrates the purpose of the `such that` notation. Consider the following goal:

```
Goal forall x:nat, (x > 1) -> (x > 0).
```

Using `Take` together with the alternative version of `Assume`, it is possible to introduce `x` and the premise in one 'sentence':

```
Take x:nat; such that x_bigger_1:(x > 1).
```

Note that this simply executes `Take` and `Assume` after that. The ';' symbol is required: it separates statements.

5.2.4 Cautions and Warnings

`Assume` can only be used if the top-level connective of the goal is an implication ($\Rightarrow$). For example, `Assume` cannot be used on a goal of the form $(A \Rightarrow B) \wedge C$.

It is highly recommended to annotate each introduced hypothesis with its type (i.e. the proposition it represents). This improves the readability of the proof, and makes it possible to break conjunctions ($\wedge$) in the premise down as desired. However, for backward compatibility, it is possible to only provide a list of hypothesis names without types.

5.2.5 Possible Errors

The possible errors are:

- `AssumeError("Cannot assume premise: goal is not an implication")`: raised if the top-level connective of the goal is not an implication ($\Rightarrow$ in mathematical notation, or `->` in Waterproof).
- `AssumeError("Too many hypotheses provided")`: raised when one tries to introduce more hypotheses than possible. For example, if the goal is `A -> B`, then the following would raise such an error, as `h_c` cannot be introduced:

  ```
  Assume h_a:A and h_c:C.
  ```

- All other variants of `AssumeError`: raised when there are more all hypotheses in the premise than stated in the `Assume` statement. For example, if the goal is `A /\ B -> C`, then

  ```
  Assume h_a:A
  ```

  will raise this error.

5.2.6 Related Operations

`Take` can also be used to create initial hypotheses for a proof, but works on goals with a $\forall$-quantifier instead.

5.3 Because

5.3.1 Functional Description

The `Because` tactic has three variations and is used for destroying a known hypothesis (within the proof of a goal) that is a conjunction or a disjunction of two statements into two respective parts.

5.3.2 Formal Description

The syntax of the first variation of the tactic is

`Because` $ident_1$ `both` $ident_2$ `and` $ident_3$.

Here, $ident_1$ is the known hypothesis, and $ident_2$ and $ident_3$ are the names of the two parts that will result after calling the respective tactic.

The syntax of the second variation of the tactic is

`Because` $ident_1$ `both` $ident_2$ `:` $constr_1$ `and` $ident_3$ `:` $constr_2$.

Here, $ident_1$ is the known hypothesis, $ident_2$ and $ident_3$ are the names of the two parts that will result after calling the respective tactic, and $constr_1$ and $constr_2$ are the *actual* statments of the two parts of the hypothesis $ident_1$, respectively.

The syntax of the third variation of the tactic is

`Because` $ident_1$ `either` $ident_2$ `or` $ident_3$.

Here, $ident_1$ is the known hypothesis, and $ident_2$ and $ident_3$ are the names of the two different cases that will result after calling the respective tactic.

5.3.3 Examples

For the first variation, an minimal example is the following:

```
Goal forall n m p: nat,
        ((n = m) /\ (m = p)) -> (n = p).
Proof.
    Take n m p : nat.
    Assume H : ((n = m) /\ (m = p)).
    Because H both H_part1 and H_part2.
```

Here, the `Take` tactic is used to introduce the variables $n, m, p \in \mathbb{N}$, and the `Assume` tactic is used to assume the premise $(n = m) \land (m = p)$ under the name of H. Then, calling the tactic

```
Because H both H_part1 and H_part2.
```

destructs H into its both parts, making the proof state contain the hypothesis

```
H_part1 : (n = m),
```

and

```
H_part2 : (m = p).
```

In the above context, instead of calling

```
Because H both H_part1 and H_part2.
```

the user could have called the second variation of the tactic, as

```
Because H both H_part1 : (n = m) and H_part2 : (m = p).
```

and the result would be the same as in the previous case.

For the third variation, consider the example:

```
Goal forall n m: nat,
        ((n = m) \/ (n + 1 = m + 1)) -> (n + 2= m + 2).
Proof.
    Take n m : nat.
    Assume H : ((n = m) \/ (n + 1 = m + 1)).
    Because H either H_part1 or H_part2.
```

Here, again, the Take tactic is used to introduce the variables $n, m, p \in \mathbb{N}$, and the Assume tactic is used to assume the premise $(n = m) \vee (n + 1 = m + 1)$ under the name of H. Then, calling the tactic

```
Because H either H_part1 or H_part2.
```

destructs H into its two different cases, making the proof state contain first contain the hypothesis

```
H_part1 : (n = m),
```

and requiring to prove the goal, then eliminating this hypothesis, introducing the hypothesis

```
H_part2 : (n + 1 = m + 1)
```

and requiring to prove the same (original) goal.

5.3.4 Cautions and Warnings

This tactic can be used also when the respective hypothesis is not a conjuction/disjunction of statements. In this case, the hypothesis will just be removed entirely from the proof, and a warning will be printed.

5.3.5 Possible Errors

None of the three tactic presented above raise any errors. However, when the tactics are used in a context in which there it is not possible to destruct a hypothesis according to these rules, because the respective hypothesis is neither a conjunction, nor a disjunction, then Coq will remove the hypothesis entirely from the proof state and print the warning that the names $ident_2$ and $ident_3$ are unused.

5.3.6 Related Operations

This tactic is not particularly related to any of the other tactics in the new tactics library.

5.4 Choose such that

5.4.1 Functional Description

The `Choose such that` tactic and is used for performing logical ∃-elimination on an existing lemma or hypothesis. That is, if it is known that a variable exists for which a specific predicate holds, then it adds this variable and the predicate to the proof state.

5.4.2 Formal Description

`Choose such that` performs ∃-elimination on an existing lemma or hypothesis. The tactic introduces the bound variable as a free variable to the proof state, and adds the predicate of the ∃-quantifier as another hypothesis. In case the tactic is applied to a hypothesis, the old hypothesis will be removed from the proof state.

Syntax The syntax of the tactic is as follows:

`Choose` $ident_1$ `such that` $ident_2$ `according to` $constr$.

Here, the $ident_1$ will become the name of the introduced free variable, and $ident_2$ will become the name of the introduced hypothesis (based on the predicate). The $constr$ should be a reference to the target hypothesis or lemma.

5.4.3 Examples

Minimal Example The following example demonstrates the syntax and semantics of the tactic. Assume that the following lemma exists and has previously been proven:

```
Lemma example_existence: exists x:nat, x + 1 = 2.
```

Then it is possible to use this lemma –within the proof of another lemma– as follows:

```
Choose x such that h according to example_existence.
```

This expression adds the following statements to the proof state:

```
x: nat
h: x + 1 = 2
```

Advanced Example As an example of a lemma to apply `Choose such that` on, consider the following convergence lemma. We say that a sequence of real numbers $(a_n)_{n\in\mathbb{N}}$ converges to the real number a if

$$\forall \varepsilon > 0, \exists N \in \mathbb{N} \text{ such that } \forall n \geq N : |a_n - a| < \varepsilon.$$

Assume this lemma has already been proven and exists in Coq under the name `conv_to` (note that this lemma takes an input, namely the limit point a).

Now an example of the usage of the tactic is the following:

```
Choose N such that an_close_to_a according (conv_to a).
```

This statement introduces the variable N from the lemma `conv_to`, and the statement

$$\forall n \geq N : |a_n - a| < \varepsilon,$$

is introduced under the name `an_close_to_a`. Note that, for the above statement to make sense, the variable ε must have already been defined.

5.4.4 Cautions and Warnings

The tactic can only be applied if the *outermost* connective of the target lemma or hypothesis is an ∃-quantifier.

The current implementation can also be abused to split a conjunctive hypothesis into two hypotheses. It is highly disrecommended to use this tactic for such a purpose, as it would confuse the reader of a proof.

5.4.5 Possible Errors

The `Choose such that` tactic does not raise any particular errors. However, when the tactic is used in a context in which there it is not possible to destruct a hypothesis according to the rules of this tactic (because there is no ∃-quantifier in the respective hypothesis), then Coq will remove the hypothesis entirely from the proof state and print the warning that the names $ident_1$ and $ident_2$ are unused.

5.4.6 Related Operations

Related to `Choose`, but this tactic eliminates an ∃ quantifier on an hypothesis. Note that this is the other way around: it uses an ∃-quantifier to introduce a new variable, whereas `Choose such that` uses an existing variable to simplify a goal.

5.5 Choose

5.5.1 Functional Description

`Choose` is a tactic that can be used to construct a proof in the reverse order. In pen-and-paper logic, one first derives a witness of a proposition, and only then introduces the ∃-quantifier as a conclusion. `Choose` allows one to introduce a *potential* witness first, and afterwards show that the proposition of the quantifier indeed holds for this witness.

5.5.2 Formal Description

The `Choose` used on an ∃ goal in order to appoint a free variable as a candidate for the bound variable. There are two variants:

- A variant that uses an existing variable as a candidate for the bound variable directly.
- A variant that uses a new variable as a candidate for the bound variable. This new variable must simply be another name for an existing variable.

Both variants practically do the same: they replace the bound variable in the goal with a free variable and removes the ∃-quantifier from the goal. It remains to prove the proposition that was originally captured in the ∃-quantifier.

Syntax There are two versions this tactic: one with a single argument, and one with two arguments

The syntax of the first version is as follows:

$$\texttt{Choose}\ (constr).$$

The argument $constr$ must be a reference to an existing free variable. This variable will be used to replace the bound variable.

The other variant has the following syntax:

$$\texttt{Choose}\ ident\ :=\ (constr).$$

The $constr$ argument must be a reference to an existing free variable, in a similar manner with the other variant. The $ident$ is the name of the newly introduced variable.

5.5.3 Examples

Simple notation A first example of the usage of this tactic is the following:

```
Lemma : forall n : nat, exists m : nat, (n = m).
    Proof.
    Take n : nat.
    Choose (n).
    This follows by reflexivity.
Qed.
```

The goal of the above lemma is to prove the statement

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N} : (n = m).$$

Here the `Take` tactic first is used to assume any free variable of the same type as the bound variable n of the ∀ goal. As a second step, the `Choose` tactic is used to pick n as a candidate for the bound variable of the ∃ quantifier. After choosing n, the remaining goal becomes $n = n$. Obviously, this holds by `reflexivity`.

Explicit notation A second example is the following:

```
Lemma : forall n : nat, exists m : nat, (n = m).
    Proof.
    Take n : nat.
    Choose m := n.
    This follows by reflexivity.
Qed.
```

Here the `Take` tactic first is used in the same way as in the previous example. Then the second variation of the `Choose` tactic is used to instantiate a free variable m that is used to replace the bound variable in the goal (note that for the free variable, another name than m would also have been valid). The new variable m is defined to be equal to n. This proof is equivalent to the previous example, but it is explicit in the instantiation of the bound variable (in the above example, the new variable is not only defined to equal n, but is also called n). After introducing m, the remaining goal is $m = n$. Since m is defined to be equal to n, this follows immediately by `reflexivity`.

5.5.4 Cautions and Warnings

The tactic can only be applied if the top-level connective of the goal is `exists` (i.e. an $\exists$-quantifier).

The tactic can be used to introduce an unsuitable variable as a witness to an $\exists$-statement, which may make it impossible to finish the proof.

5.5.5 Possible Errors

There are two possible errors:

- `ChooseError("'Choose' can only be applied to 'exists' goals")`: raised if the top-level quantifier of the goal is not `exists` (i.e. not an $\exists$-quantifier).
- `The variable` *constr* `was not found in the current environment.`: raised if the the given *constr* does not refer to an existing variable.

5.5.6 Related Operations

Related to `Choose such that`, but this tactic uses an existing variable to make progress towards proving an $\exists$-goal instead of introducing a new variable.

5.6 Contradiction tactics

5.6.1 Functional description

The `contradiction tactics` are comprised of two different tactics:

`We argue by contradiction` and `Contradiction`.

None of the above two tactics take any parameters as input.

The `We argue by contradiction` tactic is used to *start* a proof by contradiction.

The `Contradiction` tactic is used to *conclude* a proof by contradiction, once a contradiction has indeed been derived.

5.6.2 Formal description

The syntax for the `We argue by contradiction` tactic is

```
We argue by contradiction.
```

The syntax for the `Contradiction` tactic is

```
Contradiction.
```

5.6.3 Examples

This example shows a valid use of the syntax. Assume that the goal is some arbitrary statement `A`.

We can start a proof by contradiction by just writing

```
    We argue by contradiction.
```

After this, a hypothesis stating that we know ¬`A` is added to the proof state.

Once a hypothesis stating that ¬`H` is known (where H is a previously known hypothesis), one can then write

```
    Contradiction.
```

and conclude the proof by contradiction.

5.6.4 Cautions and warnings

There are no cautions and/or warning about these tactics.

5.6.5 Possible errors

The only possible error when using these tactics is when calling the `Contradiction` tactic when no contradiction has been derived. Then, the Coq error

```
No such contradiction
```

is shown.

5.6.6 Related operations

These tactics are not related to any other tactics in the new tactics library.

5.7 Define

5.7.1 Functional Description

The `Define` tactic is used to introduce a new variable within a proof, without affecting the proof state. This is equivalent to renaming an already existing variable, and also adding the renamed variable to the list of known hypotheses.

5.7.2 Formal Description

The syntax of the `Define` tactic is

`Define` *ident* : *constr*.

Here, *ident* is the name of the variable that is defined, and *constr* is the already existing variable.

5.7.3 Examples

Considering the following example:

```
Goal forall n : nat, n = n.
Proof.
    Take n : nat.
    Define m := n.
```

This example uses the `Take` tactic to introduce the bound variable of the ∀ goal as the free variable n. Then the statement

```
Define m := n.
```

introduces a new variable under the name of m and equal to n to the list of known hypotheses.

5.7.4 Cautions and Warnings

The only error that can occur is when using the `Define` tactic to rename a variable that does not exist. In this case, the tactic fails with an error.

5.7.5 Possible Errors

When using the `Define` tactic with the *constr* variable being a variable that does not exist in the proof state, Coq will raise the error

`The reference` *constr* `was not found in the current environment.`

5.7.6 Related Operations

The `Define` tactic is related to the `Take` and `Assume` tactics, which are also used for introducing variables and/or premises, but in a different context.

5.8 Either

5.8.1 Functional Description

The `Either` tactic is used to start a proof by case distinction. It can be used to create a distinction into only *two* different cases.

5.8.2 Formal Description

The syntax of the `Either` tactic is:

`Either` $constr_1$ or $constr_2$.

Here, $constr_1$ and $constr_2$ are the two cases by which the case distinction is made. Remark that $constr_2$ must be the "complement" of $constr_1$.

5.8.3 Examples

Consider the following example:

```
Goal forall x : R, exists n : nat, n > x.
Proof.
    Take x : R.
    Either (x <= 0) or (x > 0).
```

Here, the `Take` tactic was used to introduce the variable x. Then, the `Either` tactic was used to continue the proof by case distinction: either $x \leq 0$, or $x > 0$.

5.8.4 Cautions and Warnings

Whenever this tactic is used the following warning is printed: `Recommendation: Please use comments to indicate the case you are in after writing this line. This helps to keep the proof readable.`

5.8.5 Possible Errors

There are no custom errors created for this tactic. However, when calling

`Either` $constr_1$ or $constr_2$,

if $constr_2$ is not the complement of $constr_1$, then Coq raises the error

`No applicable tactic.`

5.8.6 Related Operations

This tactic is not particularly related to any of the other tactics in the new tactics library.

5.9 Help

5.9.1 Functional Description

In order to request a hint, one can use the `Help` tactic. This tactic does not change the proof state, but reads the goal and tries to print a suitable hint.

5.9.2 Formal Description

The `Help` tactic gives hints to the user on how to start the proof of the current goal. It only supports three types of goals: ∀-goals, ∃-goals, and implication-goals. In case any other goal is the topmost goal, it will only output '`No hint available.`'.

Syntax The syntax of this tactic is unique and contains no arguments. One uses this tactic as follows:

```
Help.
```

5.9.3 Examples

This section will provide for all the types of hints that `Hint` can produce.

Universal quantifier (∀) An example with a universal quantifier as a goal is the following:

```
Lemma reflexivity_of_nat: forall n : nat, (n = n).
Proof.
Help.
```

After calling the `Help` tactic, Waterproof returns the following message to the user:

```
The goal has a forall-quantifier ().
You may need to introduce an arbitrary variable of type nat.
This can for example be done using 'Take ... : ...'.
Or you may need to make an assumption stating nat.
This can for example be done using 'Assume ... : ...'.
```

Existential quantifier (∃) An example with an existential quantifier as a goal is the following:

```
Lemma particular_reflexivity_of_nat: exists n : nat, (n = n).
Proof.
    Help.
```

After calling the `Help` tactic, Waterproof returns the following message to the user:

```
The goal has an existential quantifier ().
You may want to choose a specific variable of type nat.
This can for example be done using 'Choose ... '.
```

Implication (⇒) An example with an implication as goal is the following:

```
Lemma example: (1 > 0) -> (0 < 1).
Proof.
    Help.
```

After calling the `Help` tactic, Waterproof returns the following message to the user:

```
The goal has an implication ().
You may need to assume the premise
(1 > 0).
This can for example be done using 'Assume ... : ...'.
```

Other goals An example that does not include a '∀', '∃' or a '⇒' in the goal is the following:

```
Lemma one_is_one: 1 = 1.
Proof.
    Help.
```

In this case, the goal is not of any form described from above, and so the `Help` tactic returns the message:

```
No hint available.
```

5.9.4 Cautions and Warnings

The `Help` tactic can safely be used at any point in an unfinished proof.

5.9.5 Possible Errors

The `Help` tactic never raises errors.

5.9.6 Related Operations

The `Help` tactic is related to the Take, Assume and Choose tactics, as it references them in the message it prints.

5.10 It holds that

5.10.1 Functional Description

The tactic `It holds that` can be used to prove intermediate statements (given as a $ident$: $constr$ pair). If the tactic can find enough evidence for a given statement, then it will try to add the statement as a newly named hypothesis to the current proof state. It is possible to help the tactic by providing a reference to a lemma that it should try to use.

The tactic can fail for two potential reasons: the statement is not true, or the statement is true but Waterproof fails to find a proof within a reasonable time.

5.10.2 Formal Description

The syntax for `It holds that` is:

`It holds that` $ident$: $constr$.

The $ident$ contains the name under which the statement will be added to the current proof state. The $constr$ contains the actual formulation of the statement.

The tactic will attempt to find a proof for the given `constr`. If this succeeds it will be added as a hypothesis to the current proof state.

An alternative syntax is:

By $constr$ `it holds that` $ident$: $constr$.

The function of the last two arguments is the same as in the variant described above. The extra argument (the first `constr`) must refer to a lemma, that will be used when the tactic tries to find a proof for the second `constr`.

5.10.3 Examples

Basic usage This example shows a valid use of the syntax. To prove some statement `my_statement : x = 0`, one can write:

```
It holds that my_statement : (x = 0).
```

If one has a previously proven lemma or theorem (say `my_lemma`) that may be needed to verify the statement, one can write:

```
By my_lemma it holds that my_statement : (x = 0).
```

Realistic context The following example shows a proof that for a natural number x, $x > 1 \wedge x < 3$ implies that x is even. In this context, `It holds that` can be used to infer from $x > 1 \wedge x < 3$ that $x = 2$, which is an intermediate statement to show that x is even:

```
Inductive even : nat -> Prop :=
    even0 : even 0
  | evenS : forall x:nat, even x -> even (S (S x)).

Lemma it_holds_example:
        forall x:nat, x > 1 /\ x < 3 -> even x.
Proof.
```

```
        intros x h.
        It holds that x_is_two: (x = 2).
        (* Change the goal to "even 2"*)
        rewrite x_is_two.
        (* Change the goal to "even 0"*)
        apply evenS.
        apply even0.
    Qed.
```

5.10.4 Cautions and Warnings

`It holds that` will fail with an error if the provided statement cannot be proven. This may also occur if the statement is true, but too difficult to prove with a few simple steps. In case the tactic fails while one is convinced that the statement is correct, it may be worth trying to prove several sub-statements first. These can be given as a hint to `It holds that`, making it simpler for the tactic to find a proof for the statement.

5.10.5 Possible Errors

`It holds that` will fail with the error `AutomationFailure ("Waterproof could not find a proof. If you believe the statement should hold, try making a smaller step.")`, in case Waterproof fails to find a proof for the given statement. This can occur if the statement does not hold, if a necessary lemma has not been given, or if the proof is too difficult to find within a few simple steps.

5.10.6 Related Operations

The tactic is very similar to `We conclude that`, except that it proofs a statement instead of the goal.

5.11 Rewrite equality

5.11.1 Functional Description

Waterproof offers four tactics that can be used to rewrite an expression using an equality. The equality can be given as a literal expression, it does not need to be a previously proven lemma. There are tactics to rewrite goals, and other tactics to rewrite hypotheses. If the used equality is of the form $A = B$, then these tactics find all occurrences of A in the target and replace them with B (the target is the goal or a hypothesis).

5.11.2 Formal Description

The tactics take a literal expression of the form $A = B$, where A and B can be any well-defined expression. The given expression is verified, and an error is raised if the tactic fails to do so. After verification, all occurrences of A in the target will be replaced by B. The target varies per tactic, as described below.

Syntax There are three variants of the tactic: `Rewrite using`, `Rewrite <- using` and `Write using as` and `Write goal using as`.

The simplest variant is `Rewrite using`, which adheres to the following syntax:

$$\texttt{Rewrite using}\ \mathit{constr}\ [\texttt{in}\ \mathit{ident}]^{?}.$$

The constr is the literal equality proposition used to rewrite the target. By default, the goal is used as the target. Optionally, an ident referring to a hypothesis can be given, so that this hypothesis will be rewritten instead.

Another variant is the following:

$$\texttt{Rewrite <- using}\ \mathit{constr}\ [\texttt{in}\ \mathit{ident}]^{?}.$$

The semantics are almost the same as in the variant above. The only difference is that if the constr of the form $A = B$, then all occurrences of B in the target will be replaced by A (instead of vice versa). Note that the target is the goal, unless an ident is given that identifies a hypothesis.

The third variant adds an explicit expected result:

$$\texttt{Write}\ \mathit{ident}\ \texttt{using}\ \mathit{constr}\ \texttt{as}\ \mathit{constr}.$$

The first argument (the ident) is used to refer to the target hypothesis to rewrite. The first constr is the equality proposition used for rewriting. The last constr is the expected result of the rewrite operation. An error will be raised if the rewritten target hypothesis does not equal this value.

The last variant is similar to the previous, except that it targets the goal:

$$\texttt{Write goal using}\ \mathit{constr}\ \texttt{as}\ \mathit{constr}.$$

The semantics are identical to the variant above, except that the target always is the goal.

5.11.3 Examples

Rewriting the goal The following example demonstrates how the tactic can be used to rewrite the goal:

```
Lemma rewrite_example_1: forall x y: nat,
                          5 * (x + y) = 5 * x + 5 * y.
Proof.
    intros x y.
    Rewrite using (forall n m p : nat,
                   n * (m + p) = n * m + n * p).
    reflexivity.
Qed.
```

The left-hand-side of the goal is rewritten from $5 \cdot (x + y)$ to $5 \cdot x + 5 \cdot y$. Note that the used proposition ($\forall n, m, p \in \mathbb{N} \ : \ n \cdot (m + p) = n \cdot m + n \operatorname{cot} p$) is simple enough to be verified automatically. It this example, it would also have been possible to replace `Rewrite using ...` with:

```
Write goal using (forall n m p : nat,
                  n * (m + p) = n * m + n * p)
as (5 * x + 5 * y = 5 * x + 5 * y).
```

Rewriting a hypothesis The following example demonstrates how a hypothesis can be rewritten.

```
Lemma test_rewrite_using_in_1: forall x y: nat,
    5 * (x + y) = 10 -> 2 = (x + y).
Proof.
    intros x y.
    intros h.
    Rewrite using (forall n m p : nat, n * (m + p) = n * m + n *
p) in h.
```

The tactic rewrites the hypothesis from $5 \cdot (x+y) = 10$ to $5 \cdot x + 5 \cdot y = 10$. The goal is unaffected by this rewrite. It this example, it would also have been possible to replace `Rewrite using ...` with:

```
Write h using (forall n m p : nat,
                  n * (m + p) = n * m + n * p)
as (5 * x + 5 * y = 10).
```

5.11.4 Cautions and Warnings

The given equality must be automatically prove-able. If the given equality is either a false proposition or too difficult to prove, then the tactic will fail with an error.

Note the direction of the equality. If one gives an expression of the form $A = B$, then occurrences of B in the target will not be replaced by A (unless using the variant of the tactic `Rewrite <- using`).

5.11.5 Possible Errors

There are three possible errors:

- `AutomationFailure( "Could not verify that the proposition used for the rewrite holds. You may need to prove this proposition first before rewriting others with it.")`: this error is raised when the literal equality proposition could not be verified. This can occur if this proposition is logically false, or if it too difficult to prove automatically.
- `RewriteError("Could not rewrite goal with this expression")`: this error is raised if the given literal is valid, but cannot be applied to the target. This can occur if the equality is of the form $A = B$, but the expression A does not occur in the target. This error will also be raised in case the literal proposition is not an equality.
- `RewriteError("Rewriting the hypothesis with this equality is possible, but did not produce the expected result")`: this error is raised if the result of the rewrite was different from the provided expected result. This error can only be raised by the `Write using as` and `Write goal using as` variants of the tactic.

5.11.6 Related Operations

The tactic `Rewrite inequality` can also be used to rewrite equalities. Given a chain of equalities $a_1 = a_2 = \cdots = a_n$, `Rewrite inequality` can also used to recursively rewrite the target. Furthermore, `Rewrite inequality` can also apply transitive relations on inequalities (involving $<$, $\leq$, $\geq$ or $>$). However, `Rewrite inequality` cannot be applied to a hypothesis (only to the goal), and it has no variant which can be annotated with the expected result.

5.12 Rewrite inequality

5.12.1 Functional Description

Statements of the form $a_1 < a_2 < \cdots < a_n$ (where each a_i is an expression that evaluates to a number) are commonly used in mathematical proofs. Such 'chains' of inequalities can also be written using the relations $\leq$, $=$, $>$ and $\geq$.

Waterproof allows users to write such chained inequalities by using the tactic

$$texttttRewriteinequality.$$

The 'chained' expression of inequalities is used to rewrite the goal directly. For example, if the goal is $a_1 < a_n$, then rewriting the goal using $a_1 < a_2 < \cdots < a_{n-1}$ would result in the new goal $a_{n-1} < a_n$. Note that this is a form of backward reasoning: $a_{n-1} < a_n$ (the new goal) implies that $a_1 < a_n$ (the old goal) because $a_1 < a_2 < \cdots < a_{n-1}$.

The tactic will finish the proof in case the 'chain' also includes the goal term. For example, assume that the goal is $a_1 < a_n$. Then the statement $a_1 < a_2 < \cdots < a_n$ is sufficient to prove the goal.

5.12.2 Formal Description

Let the goal be a statement of the form $a_1\ R_0\ a_n$. Then `Rewrite inequality` takes an argument of one of the following two forms:

1. $a_1\ R_1\ a_2\ R_2 \ldots R_{n-2}\ a_{n-1}$
2. $a_1\ R_1\ a_2\ R_2 \ldots R_{n-1}\ a_n$

Here $R_i \in \{<\ .\ \leq, =, \geq, >\}$ for each $i = 0, 1, \ldots, n$. The tactic will verify the correctness of each of the given (in)equalities $a_1\ R_1\ a_2$, $a_2\ R_2\ a_3$, etc. An error will be raised in case any of these (in)equalities is invalid. Let $\hat{R}$ be the logically weakest relation of $\{R_0, R_1, \ldots, R_n\}$ (where $\leq$ is weaker than $<$ which is weaker than $=$. Similarly, $\geq$ is weaker than $>$ with is weaker than $=$). Assume that there is a valid transitive relation between all given (in)equalities (if this is not the case, an error will be raised). In case the argument is of the form (1), then the goal will be rewritten to $a_{n-1}\hat{R}a_n$. In case the argument is of the form (2), then the tactic will prove the goal.

Syntax The formal syntax of the tactic is as follows:

`Rewrite inequality` $constr$ $[$`"<"`$|$`"<="`$|$`"≤"`$|$`"="`$|$`">="`$|$`"≥"`$|$`">"`$]$, $constr]^+$.

Each $constr$ is a mathematical expression that evaluates to a number (or any other mathematical object for which the (in)equality relations are defined). The leftmost $constr$ must be the left-hand-side term in the goal. The goal will be completely proven in case that the last $constr$ is the goal's right hand side. Otherwise, the last $constr$ will become the new left hand side of the goal. Mind that the quotes surrounding the (in)equality signs are necessary. `"<="` and `">="` are alternative notations for `"≤"` and `"≥"` respectively [5].

5.12.3 Examples

This section first gives two basic examples: one for each type of input (as described in Section 5.12.2). A more advanced example follows, that shows that different (in)equality relations can be used in a single chain.

[5] These notations are included because the symbols $\leq$ and $\geq$ might be difficult to type on a keyboard.

Rewriting a goal The following is a minimal example of using `Rewrite inequality` to rewrite a goal:

```
Lemma example_rewrite_1: (3 >= 2) -> (3 >= 1).
intros h.
Rewrite inequality 3 " " 2.
```

The resulting goal will be $2 >= 1$. Note that the rightmost expression in the input chain (i.e. 2), does not equal the right-hand side of the goal (i.e. 1).

Finishing a goal The following example shows how `Rewrite inequality` can be used to prove a goal:

```
Lemma example_rewrite_2: forall a b c: R, (a < b < c) -> (a < c)
.
    intros a b c h.
    destruct h as [h1 h2].
    Rewrite inequality a "<" b "<" c.
Qed.
```

At the point where `Rewrite inequality` is called, the hypotheses `a < b` and `b < c` exist. This ensures that the tactic will be able to verify each individual inequality. The tactic proved the goal, since the last expression int eh input 'chain' `a "<" b "<" c` is the right-hand side of the goal (`C`).

Different relations The following example shows that the inequality '$<$' can be used in the same chain as '$=$':

```
Lemma example_rewrite_3:
    forall a b c d: R, (a < b /\ b = c /\ c < d) -> (a < d).
    intros a b c d h.
    destruct h as [h1 h2].
    destruct h2 as [h2 h3].
    Rewrite inequality a "<" b "=" c "<" d.
Qed.
```

5.12.4 Cautions and Warnings

`Rewrite inequality` can only be used if each individual (in)equality in the provided 'chain' is logically valid.

The complete chain supplied to `Rewrite inequality` must be logically valid as well (by use of transitivity). In practice, this means that chains that include $<$ and/or $\leq$ cannot include $>$ and/or $\geq$ (or vice versa). For example, take the chain $A < B > C < D$. This chain cannot be used to prove that $C < D$ implies $A < D$, even if each of $A < B$, $B > C$ and $C < D$ is logically valid.

The inequality signs used in the syntax must be surrounded by double quotes (`""`)[6].

[6]This is due to an unfortunate limitation of the language used to implement the tactics.

5.12.5 Possible Errors

- `AutomationFailure("Could not verify that the proposition used for the rewrite holds. You may need to prove this proposition first before rewriting others with it.")`: this error is raised in case any individual inequality of the input is logically invalid, or too difficult to verify automatically.
- `RewriteError("")`: this error will also occur if the chain is not connected by transitive relations (i.e. contains a variant of 'greater than' and a variant of 'smaller than'). A message will be printed that explains with (in)equality symbol is invalid, and at which point, during the rewrite, the error occurred.
- `RewriteError("Invalid (in)equality connective")`: this error is raised in case of a syntactic mistake: one (in)equality symbol is not supported. The supported symbols are $<$. $\leq$, $=$, $\geq$, and $>$.
- `RewriteError("LHS of (in)equality must match LHS of goal.")`: this error is raised if a correct (in)equality cannot be applied to the goal, but the leftmost term (a_1) in the 'chain' is not equal to the left-hand-side term of the goal.

5.12.6 Related Operations

The tactic can be seen as a recursive version of `Rewrite equality`. However, there are other differences:

1. `Rewrite equality` can also be applied to an hypothesis.
2. `Rewrite equality` has a variant (`Write using` and `Write goal using`) that check if the result of the rewrite is as expected.
3. `Rewrite inequality` supports inequality relations, `Rewrite equality` supports only equalities.
4. `Rewrite inequality` can also prove a goal, depending on the input argument.

5.13 Sets tactics

5.13.1 Functional description

The `sets tactics` are comprised of two different tactics:

`We prove equality by proving two set inclusions`

and

`This set equality is trivial.`

None of the above two tactics take any parameters as input.

The `We prove equality by proving two set inclusions` tactic is used when the goal is of the form $A = B$ where A and B are sets. This tactic then destructs the goal into two separate sub-goals: proving that $A \subset B$ and proving that $B \subset A$ (this is equivalent to showing that $A = B$, due to the axiom of extensionality).

The `This set equality is trivial` is an automation tactic specifically designed for proving "trivial" set equalities, of the form $A \cap B = B \cap A$, $A \cap \emptyset = \emptyset$ etc.

5.13.2 Formal description

The syntax for the `We prove equality by proving two set inclusions` tactic is

`We prove equality by proving two set inclusions.`

The syntax for the `This set equality is trivial` tactic is

`This set equality is trivial.`

None of the above tactics take any input parameters.

5.13.3 Examples

This example shows a valid use of the syntax.

For the `We prove equality by proving two set inclusions`, assume that the goal is some arbitrary statement of the form `A = B`, where A and B are sets. Calling the respective tactic is done as follows:

```
We prove equality by proving two set inclusions.
```

After this, the goal is destructed into two different goals $A \subset B$ and $B \subset A$.

For the `This set equality is trivial`, assume that the goal is a "trivial" set equality, for instance, $A \cap B = B \cap A$. Calling the respective tactic is done as follows:

```
This set equality is trivial.
```

After this, the automation function finishes the proof of the goal.

5.13.4 Cautions and warnings

Even though the `This set equality is trivial` tactic had a specific design scope in mind, it can of course be used in more involved set equalities. However, it is very likely that it will return an error when used in such situations.

5.13.5 Possible errors

When calling the `This set equality is trivial` tactic, only one error can occur: when the automation is not able to finish the proof of the goal. The error that is shown is the same error as for the We conclude that tactic.

5.13.6 Related operations

These tactics are not related to any other tactics in the new tactics library.

5.14 Simplify what we need to show

5.14.1 Functional Description

The `Simplify what we need to show tactic` is used to perform straightforward simplifications in the current goal, in a proof. If there are no simplifications to be made, the tactic does not change anything.

5.14.2 Formal Description

The syntax of this tactic is

```
Simplify what we need to show.
```

Remark that this tactic does not take any input parameters.

5.14.3 Examples

Consider the following example:

```
Goal forall a b : nat, (a, b).2 = (a, b).2.
Proof.
    Take a, b : nat.
    Simplify what we need to show.
```

Here, the goal is to prove that, for every $a, b \in \mathbb{N}$, the second entry in the pair of elements (a, b) is equal to itself. After using the `Take` tactic to introduce the variables a and b, the tactic

```
Simplify what we need to show.
```

is called. This simplifies the goal, by transforming it into proving that

```
b = b.
```

5.14.4 Cautions and Warnings

There are no cautions/warnings for this tactic. In case it is used where not applicable, the tactic will simply have no effect on the proof state.

5.14.5 Possible Errors

The tactic `Simplify what we need to show` does not raise any errors.

5.14.6 Related Operations

The tactic `Unfold` can be used to replace a function by its definition. Unlike `Simplify what we need to show`, `Unfold` never attempt to also evaluate the function.

5.15 Take

5.15.1 Functional Description

The tactic `Take` is used to start the proof of a ∀-goal. It introduces free variables of the type of the bound variable of the quantifier. The user should proceed in proving the body of the quantifier.

`Take` can introduce multiple variables in a single command, in case the goal contains a ∀-quantifier over multiple variables.

The most basic syntax to introduce a variable with identifier (name) "`x`" of type `T` is:

```
Take x : T.
```

One can introduce multiple variables of the same type by providing multiple identifiers, each separated by a comma and followed by a single type. For example, to introduce three variables "`x`", "`y`" and "`z`", all of type `T`:

```
Take x, y, z : T.
```

This can be extended to an arbitrary amount of identifiers. Furthermore, multiple identifiers-type lists can be introduced at the same time. For example, to introduce two variables "`x`" and "`y`" of type `T`, *and* variables "`z`" and "`a`" of type `V`:

```
Take x, y : T, a, z : V.
```

5.15.2 Formal Description

The formal definition of the syntax is:

$$\texttt{Take}\ [[ident]^{+}_{,} : constr]^{+}_{,}.$$

5.15.3 Examples

A minimal example showing the usage of `Take` is the following:

```
Lemma n_is_n: forall n:nat, n = n.
Proof.
    Take x:nat.
    reflexivity.
Qed.
```

After executing the line

```
Take x : nat.
```

the goal becomes `x = x`. Note that the bound variable `n` is replaced with the concrete variable `x`.

5.15.4 Cautions and Warnings

The `Take` tactic can only be used if the top-level connective of the goal is a ∀-quantifier. Note that, for example, the goal $(\forall x \in A\ :\ x > 0) \Rightarrow (0 \notin A)$ has a '⇒' as top-level connective, not a '∀'.

The tactic requires the type of each variable to be annotated. For example, if one introduces a variable x while the goal is $\forall x \in \mathbb{N} \ : \ x \geq 0$, then it is required to annotate x as being of type `nat`.

$\forall$-quantifiers over multiple types of bound variables must be eliminated in the correct order. This is because such a quantifier is internally represented as nested quantifiers, so the types of the variables must match the order of the types of the bound variables. For example, given:

```
Lemma bool_and_nat: forall (n: nat)  (b : bool), n < S n = eqb b
b.
```

This is internally represented as:

$$\forall n \in \mathbb{N}, \forall b \in \{0, 1\} \ : \ (n < n + 1) = (b = b).$$

Hence, the following is ill-nested:

```
Take b : bool, n : nat.
```

Instead, one can use:

```
Take n : nat, b : bool.
```

5.15.5 Possible Errors

There are two possible errors:

- `TakeError("The type of the variable must match the type of the 'forall' goal's bound variable.")`: this error is raised when the type of the introduced variable is different from the type of the bound variable of the $\forall$-quantifier. This error will also be raised if the goal is a $\forall$-quantifier over multiple variables of different types, and they are introduced in a different order than as in the goal.
- `TakeError("'Take' can only be applied to 'forall' goals")`: raised if the top-level quantifier of the goal is not `forall` (i.e. not a $\forall$-quantifier).

5.15.6 Related Operations

The `Assume` tactic provides similar functionality, but instead of assuming variables it assumes premises of a an implication goal (a goal whose top-level connective is '$\Rightarrow$').

5.16 This follows by assumption

5.16.1 Functional Description

The `This follows by assumption` and `Then holds by assumption` tactics simply call the Coq `assumption` tactic, finishing a proof where the goal is already known as a hypothesis in the proof state.

5.16.2 Formal Description

The syntax of the `This follows by assumption` tactic is

`This follows by assumption.`

Note that this tactic does not take any input parameters.

The syntax of the `Then holds by assumption` tactic is

`Then` *constr* `holds by assumption.`

Here, the *constr* must be the current goal. In case the given *constr* is logically equivalent to the actual but not identical (i.e. equal after rewriting), a warning will be printed, but the statement will be executed. An error will be raised in case the provided and actual goal are logically different.

5.16.3 Examples

Consider the following example:

```
Goal forall n m : nat, (n = m) -> (n = m).
Proof.
    Take n, m : nat.
    Assume H : (n = m).
    This follows by assumption.
```

Here, the `Take` tactic is used to introduce the variables $n, m \in \mathbb{N}$, and the `Assume` tactic is used to assume the premise that $(n = m)$ (this is equivalent to `intros n m h`). Then, calling the tactic

```
This follows by assumption.
```

finishes the proof, as there is a hypothesis stating that $n = m$ (called H). Alternatively, the tactic

```
Then (n = m) follows by assumption.
```

could have been used instead of the `This follows by assumption` tactic, yielding the same output as before.

5.16.4 Cautions and Warnings

When calling the `This follows by assumption` tactic and the current proof state does not contain a hypothesis that is directly equal to the current goal, then the tactic will fail with an error.

A warning will be raised when calling the `Then holds by assumption` tactic in case the parameter given to the tactic is logically equivalent to the current goal, but not directly the same. In this case, Waterproof will rewrite the goal and proceed with proving it.

5.16.5 Possible Errors

There are three possible situations that can give rise to errors/warnings:

- If the user calls the `This follows by assumption` tactic or its second variation, and there is no assumption that is exactly equal to the current goal, then Coq raises the error

  ```
  No such assumption.
  ```

- If the user calls the `Then holds by assumption` tactic, and its parameter is not logically equivalent to the current goal, then the error

  ```
  GoalCheckError("No such goal").
  ```

  is raised.

- If the tactic `Then holds by assumption` is called, and the given parameter is not equivalent to the currently active goal (only a warning will be given in case the given goal is equivalent to the actual goal, but only equal after rewriting the actual goal).

5.16.6 Related Operations

The We conclude that tactics can apply the Coq `assumption` tactic as well, making them a substitute for the `This follows by assumption` tactics.

5.17 This follows by reflexivity

5.17.1 Functional Description

The `This follows by reflexivity` tactic simply calls the Coq `reflexivity` tactic, finishing a proof where the goal is of the form $A = A$, for some mathematical object A.

5.17.2 Formal Description

The syntax of this tactic is

```
This follows by reflexivity.
```

Note that this tactic does not take any input parameters.

5.17.3 Examples

Consider the following example:

```
Goal forall n : nat, n = n.
Proof.
    Take n : nat.
    This follows by reflexivity.
```

Here, the `Take` tactic is used to introduce the variable $n \in \mathbb{N}$. Then, calling the tactic

```
This follows by reflexivity.
```

finishes the proof, as the current goal is to prove that $n = n$.

5.17.4 Cautions and Warnings

If the `reflexivity` tactic cannot be applied (because the goal is not of the form $A = A$ for some mathematical object A), then the `This follows by reflexivity` tactic fails with an error.

5.17.5 Possible Errors

If the goal is not of the form $A = A$ for some mathematical object A, and the `This follows by reflexivity` tactic is called, then the error

```
ReflexivityError("Reflexivity cannot be applied here.")
```

is raised.

5.17.6 Related Operations

The We conclude that tactics can apply the Coq `reflexivity` tactic as well, making them a substitute for the `This follows by reflexivity` tactic.

5.18 Unfold

5.18.1 Functional Description

The `Unfold` and `Expand the definition of` tactics are used to expand the definition of a previous result in either a hypothesis inside the proof state, or in the current goal.

5.18.2 Formal Description

There are two variations for each tactic: one that expands the definition of a previous result inside a hypothesis, and one that does the same, but in the current goal.

For the former, the syntax of each tactic is then

$$\texttt{Unfold}\ \mathit{constr}\ \texttt{in}\ \mathit{ident},$$

and

$$\texttt{Expand the definition of}\ \mathit{constr}\ \texttt{in}\ \mathit{ident}.$$

Here, constr is the name of a previously proven statement or a previously defined hypothesis, and ident is the name of the hypothesis in which the definition of the statement is expanded.

For the latter, the syntax of each tactic is then

$$\texttt{Unfold}\ \mathit{constr},$$

and

$$\texttt{Expand the definition of}\ \mathit{constr}.$$

Here, constr is the name of a previously proven statement or a previously defined hypothesis, whose definition is expanded in the current goal.

5.18.3 Examples

Assume that we have a definition called `is_bounded`, defined as:

$$\texttt{is\_bounded}\ A : \left[\exists M > 0, \forall x : A, |x| < M\right],$$

where $A \subset \mathbb{R}$.

For the first variation, consider the following example:

```
Goal is_bounded A -> (exists M > 0, forall x : A, |x| < M).
Proof.
    intros H.
    Expand the definition of is_bounded in H.
```

Here, the `intros` tactic was used to introduce the premise `is_bounded A` under the name of H (cf. 5.2). When calling the tactic

```
Expand the definition of is_bounded in H.
```

the definition of `is_bounded` is expanded in H, yielding the changed hypothesis:

$$H : \left[\exists M > 0, \forall x : A, |x| < M\right].$$

One could instead call the tactic

```
    Unfold is_bounded in H.
```

and this tactic has the same behaviour.

For the second variation, consider the following example:

```
    Goal is_bounded A.
        Proof.
        Expand the definition of is_bounded.
```

Here, the goal is to prove `is_bounded A`. After calling the tactic

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

the goal now becomes

$$\exists M > 0, \forall x : A, |x| < M.$$

5.18.4 Cautions and Warnings

These tactics can safely be used at any point throughout the course of a proof. When the tactic is used where not applicable, the proof state will simply not be changed.

5.18.5 Possible Errors

None of the previously mentioned tactics raise any errors.

5.18.6 Related Operations

The tactic `Simplify what we need to show` can also be used to rewrite functions. However, this tactic will also attempt to evaluate functions (e.g. it evaluates a `match` statement in a function if possible). `Unfold` never attempt to also evaluate a function.

5.19 We claim that

5.19.1 Functional Description

The tactic `We claim that` is used to add a new sub-goal throughout the course of a proof. This sub-goal is added the top-most goal in the proof. After this particular sub-goal is proven, it is added as a hypothesis to the proof state.

5.19.2 Formal Description

The syntax of the `We claim that` tactic is

`We claim that` *ident* : *constr*.

Here, *constr* is a statement that the user intends to prove. After this statement is proven, it is added as a hypothesis to the proof state, and the name of the hypothesis stating this result will be *ident*.

5.19.3 Examples

Consider the following example:

```
Goal forall n : nat, exists m : nat, n = m.
    Proof.
    We claim that H : (forall k : nat, k = k).
    Take n : nat.
    reflexivity.
```

In the above example, there is a single goal: to prove that

$$\forall n \in \mathbb{N}, \exists m \in \mathbb{N}, m = n.$$

After calling the tactic

```
We claim that H : (forall k : nat, k = k).
```

a new sub-goal is added as the top-most goal in the proof state: proving that

$$\forall n \in \mathbb{N}, n = n.$$

Then, after calling the tactics (one of which being the `Take` tactic, to introduce the variable $n \in \mathbb{N}$)

```
Take n : nat.
reflexivity.
```

the newly added claim is proven, and added as a hypothesis to the proof state, under the name of H.

5.19.4 Cautions and Warnings

There are no warnings with respect to applying this tactic.

5.19.5 Possible Errors

There `We claim that` tactic does not raise any errors: any statement can be asserted to be proven.

5.19.6 Related Operations

The `We claim that` tactic is the first part of the functionality of the `It holds that` tactic, which also asserts (but also automatically tries to prove) a new claim, and then adds it as a hypothesis.

5.20 We conclude that

5.20.1 Functional Description

The tactic `We conclude that` can be used to finish the proof of a goal. Similarly to `It holds that`, it will try to automatically find a proof for the goal.

One explicitly needs to write down the goal that is to be concluded. This feature enforces better readability of the code.

The tactic can fail for three potential reasons: the statement is not true, the statement is true but Waterproof fails to find a proof within a reasonable time, or the provided goal does not match the actual goal.

Variants This tactic has a variant, `By ... we conclude that`, which passes an additional lemma as a hint to the automatic proof resolution. This version is useful if one knows that the goal follows from a specific lemma, but the tactic fails without supplying it.

Alternatively, there are also two variants of this tactic that do not require to be annotated with the goal (nor take an additional hint). This are `This follows immediately` and `This concludes the proof`.

5.20.2 Formal Description

The syntax for `We conclude that` is:

`We conclude that` *constr*.

or the equivalent alternative:

`It follows that` *constr*.

Here the *constr* must be the proposition that has been derived. The tactic will only print a warning in case the *constr* is logically equivalent to the actual but not identical (i.e. only equal after rewriting). It will still execute the automation in this case. The tactic raises an error in case the provided and actual goal are logically different.

If the tactic succeeds to find a proof for the given goal, the goal will be removed from the proof state. In case the tactic fails to find a proof an error will be raised.

Extended variant An extended syntax is available, which includes a hint for the automatic proof completion. This similar to the alternative syntax for `It holds that`:

`By` *constr* `we conclude that` *constr*.

The function of the last two arguments is the same as in the variant described above. The extra argument (the first `constr`) must refer to a lemma, that will be used when the tactic tries to find a proof for the second `constr`.

Simple variants The syntaxes for the variants that do not take any argument are:

```
This follows immediately.
```

and:

```
This concludes the proof.
```

These variants behave the same as `We conclude that`, except that they do not take a *constr* argument that should represent the goal. `This concludes the proof` also print a warning that recommends to write the goal explicitly using `We conclude that`.

5.20.3 Examples

Basic usage This example shows a valid use of the syntax. Assume that the goal is some arbitrary statement `(A)`, for which enough evidence has been derived. Then one can write:

```
    We conclude that A.
```

If one has a previously proven lemma or theorem (say `my_lemma`) that may be needed to proof the goal, one can write:

```
    By my_lemma we conclude that A.
```

Realistic context The following example shows how a specific lemma (actually the definition `even0`) can be used to conclude a proof. The lemma states that all natural numbers equal to 2 are even[7].

```
    Inductive even : nat -> Prop :=
        even0 : even 0
      | evenS : forall x:nat, even x -> even (S (S x)).

    Lemma sum_example_by_we_conclude: forall x:nat, x = 2 -> even x.
    Proof.
        intros x h.
        rewrite h. (* Change the goal to "even 2"*)
        apply evenS. (* Change the goal to "even 0"*)
        By even0 we conclude that (even 0).
Qed.
```

Note that the goal changed during the proof, so the goal stated in the last line (`even 0`) does not equal the statement of the lemma (`forall x:nat, x = 2 -> even x`). This makes sense, as `even 0` is the only statement that is still required to conclude the proof (already an arbitrary $x \in \mathbb{R}$ been assumed, it is shown that if 2 is even, then also 0 is even (by definition of `evenS`), etc.).

5.20.4 Cautions and Warnings

Similarly to `It holds that`, `We conclude that` will fail with an error if the current goal cannot be proven. Assuming that the goal is a true statement, one may need to provide a different lemma, or prove intermediate steps first.

`We conclude that` will also fail with an error if a logically different goal has been written down than the actual currently active goal.

A warning will be raised in case a goal is given that is logically equivalent to the actual goal, but not directly the same. In this case, Waterproof will rewrite the goal and proceed with proving it.

5.20.5 Possible Errors

`We conclude that` will fail with an `AutomationFailure`-error in two cases:

[7] This may sound a bit peculiar, since there is only one natural number equal to 2. However, logically this is a sound proposition.

- `AutomationFailure("Given goal not equivalent to actual goal.")`: this error is raised if the provided goal is logically not equivalent to the topmost goal of the proof state. This only applies to the variants `We conclude that` and `By ... we conclude that`.
- `AutomationFailure("Waterproof could not find a proof. If you believe the statement should hold, try making a smaller step.")`: this error is raised in case the automation features fail to prove the topmost goal. The goal might either be too difficult to prove, or may logically be false[8]. This may also occur if the necessary lemma was not given (as argument to `By ... we conclude that`).

5.20.6 Related Operations

The tactic is very similar to `It holds that`, except that it proofs the goal instead of an arbitrary statement.

[8] Or logically unproveable (i.e. undecidable).

5.21 We know

5.21.1 Functional Description

The `We know` and the `By we know` tactics are used to improve the readability of a proof, by reminding the user and the reader of the proof of an already existing hypothesis.

5.21.2 Formal Description

The syntax of the `We know` tactic is

`We know`$ident$ `:` $constr$.

and the syntax of the `By we know` tactic is

`By` $ident$ `we know` $constr$.

Here, $constr$ is the hypothesis that is known to be true, and $ident$ is the name under which the respective hypothesis is saved in the proof state.

5.21.3 Examples

Consider the following example:

```
Goal forall n m : nat,
        (n = m) -> (n + 1 = m + 1).
Proof.
    Take n m : nat.
    Assume H : (n = m).
    We know H : (n = m).
    By H we know (n = m).
```

Here, the `Take` tactic was used to introduce the variables $n, m \in \mathbb{N}$, and the `Assume` tactic was used to assume the premise $n = m$ under the name of H.

Then, calling the tactic

```
We know H : (n = m).
```

prints the message

```
We indeed know that (n = m),
```

and calling the tactic

```
By H we know (n = m).
```

prints the same message.

5.21.4 Cautions and Warnings

If any of these two tactics is used to remind the user of a hypothesis that does not exist or is not true, then the tactic fails with an error.

5.21.5 Possible Errors

If any of these two tactics are called with a $constr$ and/or an $ident$ that is not known/saved in the proof state until that point, then the tactics raise the error

```
WeKnowError("This hypothesis does not exist.").
```

5.21.6 Related Operations

These tactics are related to the We need to show tactic, which has the purpose of reminding the user/reader of the goal that needs to be proven.

5.22 We need to show

5.22.1 Functional Description

The `We need to show` tactic, together with its other seven variations, are used to improve the readability of a proof, by reminding the user and the reader of the goal that needs to be proven.

5.22.2 Formal Description

The syntax of each variation is the following:

`We need to show` *constr*,

`We need to show :` *constr*,

`We need to show that` *constr*,

`We need to show that :` *constr*,

`To show` *constr*,

`To show :` *constr*,

`To show that` *constr*,

and

`To show that :` *constr*.

Here, *constr* is the statement of the goal that needs to be proven.

5.22.3 Examples

Consider the following example:

```
Goal 1 = 1.
    Proof.
    We need to show (1 = 1).
    We need to show that (1 = 1).
    We need to show : (1 = 1).
    We need to show that : (1 = 1).
    To show (1 = 1).
    To show that (1 = 1).
    To show that : (1 = 1).
    To show : (1 = 1).
```

Here, calling each variation of the `We need to show tactic` prints the same message:

```
The goal is indeed : 1 = 1.
```

5.22.4 Cautions and Warnings

If one of these tactics is used to remind the user of the goal that needs to be proven, but the parameter passed to this tactic is not logically the same as the actual goal, then each tactic fails with an error.

5.22.5 Possible Errors

If any of these tactics is called with the $constr$ parameter not being logically the same as the actual goal, then each tactic raises the error

```
GoalCheckError("No such goal").
```

5.22.6 Related Operations

These tactics are related to the We know tactics, which have the purpose of reminding the user/reader of a known hypothesis within the proof state.

5.23 We prove by induction on

5.23.1 Functional Description

The `We prove by induction on` tactic is used to start a proof by mathematical induction in a goal starting with a ∀-quantifier.

5.23.2 Formal Description

The syntax of this tactic is

`We prove by induction on` $ident$,

where $ident$ is the name of the variable on which the induction wants to be performed.

5.23.3 Examples

Consider the following example:

```
Goal forall n : nat, n < 2*n.
    Proof.
   intros n.
    We prove by induction on n.
```

A proof by mathematical induction on n starts after calling the line

```
We prove by induction on n.
```

Coq then requires the user to prove the *base case* first, and then the *induction step*. Note Coq will automatically add the *induction hypothesis* to the proof state after finishing the base case.

5.23.4 Cautions and Warnings

Due to the nature of Coq (which uses a foundational system of Mathematics called *type theory* and not set theory), one can apply this tactic on any variable bound by a ∀-quantifier. However, caution is advised, as these types of induction are not the same as the mathematical induction in set-theoretical Mathematics.

5.23.5 Possible Errors

There are no customised errors for this tactic. Moreover, as previously mentioned, it is technically possible to apply induction to any variable bound by a ∀-quantifier.

However, if one tries to apply the `We prove by induction on` tactic to a variable that is not bound by a ∀-quantifier, the Coq raises the error

`No such assumption.`

5.23.6 Related Operations

This tactic is not related to any other tactics in the tactics library.

5.24 We show/prove both directions

5.24.1 Functional Description

The `We show both directions` and `We prove both directions` tactics are used when proving an if-and-only-if goal, i.e. a goal that is an equivalence of two statements.

These two tactics split the equivalence into its two respective directions, creating a separate goal for each part.

5.24.2 Formal Description

The syntax of the `We shot both directions` tactic is

```
We show both directions.
```

whereas the syntax of the `We prove both directions` tactic is

```
We prove both directions.
```

5.24.3 Examples

A minimal example is the following:

```
Goal forall n, m : nat,
        (n = m) <-> (n + 1 = m + 1).
    Proof.
    Take n, m : nat.
```

Calling the `Take` tactic introduces the variables $n, m \in \mathbb{N}$. The goal is now to prove the statement

$$(n = m) \iff (n + 1 = m + 1).$$

After calling the `We show both directions` or the `We prove both directions` tactics, as

```
We show both directions.
```

or

```
We prove both directions.
```

the goal is split into two parts, and the user is now required to prove the following two statements:

$$(n = m) \Rightarrow (n + 1 = m + 1),$$

and

$$(n + 1 = m + 1) \Rightarrow (n = m).$$

5.24.4 Cautions and Warnings

These two tactics will fail with an error if the current goal is not an equivalence (an if-and-only-if statement).

5.24.5 Possible Errors

There is only one error that can occur when calling these two tactics: when the current goal is not an equivalence of two separate statements, the tactics will raise the error

```
BothDirectionsError("This is not an if and only if, so try another tactic.")
```

5.24.6 Related Operations

The `We show both statements` tactic is similar to the `We show both directions tactic`, but it is instead used when the goal is a conjunction of statements.

5.25 We show both statements

5.25.1 Functional Description

The `We show both statements` tactic (and its other two variations) are used when proving a conjunction-goal, i.e. a goal that is a conjunction of two propositions.

These tactics split the conjunction into its two respective parts, creating a separate goal for each part.

5.25.2 Formal Description

There are three variations of this tactic. Two of them have no arguments, and one of them has arguments.

The syntax of the first variation of the tactic is

`We show both statements.`

The syntax of the second variation of the tactic is

`We prove both statements.`

The syntax of the third variation of the tactic is

`We prove both` $constr_1$ `and` $constr_2$.

Here the $constr_1$ and $constr_2$ must be the propositions that make up the goal, i.e. for which the goal is of the form

$$constr_1 \wedge constr_2 \text{ or } constr_2 \wedge constr_1.$$

5.25.3 Examples

A minimal example is the following:

```
Lemma : forall n : nat, (n = n) /\ (n + 1 = n + 1).
```

The goal of the above lemma is to prove the statement

$$\forall n \in \mathbb{N} : (n = n) \wedge (n + 1 = n + 1).$$

After calling the `We show both statements` or the `We prove both statements` tactics, as

```
We show both statements.
```

or

```
We prove both statements.
```

the goal is split into two parts, and the user is now required to prove the following two statements:

$$\forall n \in \mathbb{N} : (n = n),$$

and

$$\forall n \in \mathbb{N} : (n + 1 = n + 1).$$

Alternatively, the user may call the tactic as

```
    We show both (n = n) and (n + 1 = n + 1).
```

or

```
    We show both (n+1 = n+1) and (n = n).
```

and the goal is also split into the respective two parts.

5.25.4 Cautions and Warnings

All three of these tactics fail with an error if the goal is not a conjunction of two statements.

One particular variation fails with a warning or with an error, depending on the context in which it is used. We will explain this in Section 5.25.5.

5.25.5 Possible Errors

For the variations `We show both statements` and `We prove both statements`, there is only one error that can occur: when the current goal is not a conjunction of two separate statements, the tactics will raise the error

```
BothStatementsError("This is not an 'and' statement, so try another tactic.")
```

For the last variation, there is a single error that can be displayed: if the goal is a conjunction of two statements, A and B, and the `We show both C and D` tactic is called, with parameters C and D for which $C, D \notin \{A, B\}$, then the error

```
BothStatementsError("None of these two statements are what you need to show.")
```

is raised. If only one of the statements A or B is not the correct one to show, a warning is displayed. For the sake of example, let us assume that the user called the tactic

```
    We show both A and C.
```

with $C \neq B$. In this case, the warning

```
You need to show B instead of C
```

is shown.

5.25.6 Related Operations

The `We show both directions` is a tactic similar to these tactics, but it is instead used when the goal is an equivalence of two statements.

5.26 Write as

5.26.1 Functional Description

The tactic `Write as` can be used to rewrite a hypothesis directly into another (logically equivalent) form. This tactic automatically tries to find a logically valid way to perform the rewrite. However, this automation is typically only able to solve trivial equivalences.

5.26.2 Formal Description

`Write as` rewrites a hypothesis into a specified form that is equivalent *by definition*. The tactic will raise an error if the old form cannot be converted to the specified form by applying only definitions.

Syntax The tactic can be used as follows:

`Write` *ident* `as` *constr*.

Here the *ident* refers to the target hypothesis to rewrite. The *constr* is the desired form to which the hypothesis will be rewritten.

5.26.3 Examples

The following example shows how `Write as` can be used in a simple context. Note that in Coq's type-theoretic system, 3 is *by definition* equivalent to $1 + 1 + 1$.

```
Lemma example: forall x, x = 1 + 1 + 1 -> x = 3.
Proof.
    intros x h.
    Write h as (x = 3).
    assumption.
Qed.
```

Originally, the hypothesis is defined as `h := 1 + 1 + 1`. The `Write as` tactic is used to rewrite the hypothesis to `h := 3`.

5.26.4 Cautions and Warnings

The tactic is very implicit in the logical steps taken. A user must be careful not to obscure the proof script by using this tactic for a non-trivial rewrite (if readability is an objective).

The tactic can only rewrite hypothesis into forms that are equivalent *by definition*. It cannot be used for more advanced equivalences, that for example rely on a lemma.

5.26.5 Possible Errors

This tactic can only raise one possible error:
`RewriteError("Cannot rewrite the hypothesis with this term.")`. This error is raised if the specified form is not *by definition* equivalent to the target hypothesis' original form.

5.26.6 Related Operations

The tactic `Rewrite equality` can also be used to rewrite a hypothesis. However, this tactic is able to find logical equivalences that are not only based on definitions (such as by using lemmas). The main advantage of `Write as` is that it is faster to execute and more concise in a proof script.

Part II
Waterproof Installer

6 Introduction

6.1 Intended readership

This document is intended for two types of users of the Waterproof software.

1. **New users:** users who are new to Waterproof and want to install the software.
2. **Existing users:** users who want to update Waterproof or delete its dependencies.

The Waterproof installer user manual assumes a basic understanding of the Windows operating system, namely the ability to download files, and browse folders is assumed.

6.2 How to use this part of the manual

This part of the manual will contain two main sections: tutorials and error messages with recovery procedures. No other references are needed, since the tutorials cover the complete functionality due to its simplicity.

6.3 Purpose

The Waterproof application [9] relies on external dependencies. The purpose of this chapter is to show how to install these Waterproof dependencies. It will also show how to update and uninstall Waterproof.

7 Overview

The following section will contain the tutorials to install and uninstall the Waterproof dependencies as well as to update Waterproof.

8 Tutorials

8.1 Install Waterproof dependencies

8.1.1 Functional description

The tutorial explains how to install the Waterproof dependencies.

8.1.2 Cautions and warnings

The Waterproof dependencies must not be already installed. If an installation is already present, uninstall it as explained in the uninstallation tutorial in Section 8.2

8.1.3 Procedure:

1. Navigate to the Releases page to download the dependencies installer at

   `https://github.com/impermeable/waterproof-dependencies-installer`.

2. Execute the installer.
3. Press yes to the pop up saying 'Do you allow this app from an unknown publisher to make changes to your account'.
4. Follow through the windows depicted in 1.
   (a) Click 'next' on the welcome screen.
   (b) Click 'I agree' to agree to the license agreement.
   (c) Select packages not needed or needed and click 'next'.
   (d) Click 'next' to install it in the default installation location.
   (e) Click 'install' to install the software.
5. The Waterproof dependencies will now be installed.

8.1.4 Possible errors

The prompt 'Windows protected your PC.' can be shown when executing the installer. This is a Windows virus protection mechanism to discourage the execution of unknown software. This can safely be ignored by clicking 'More info' followed by 'Run anyway'.

If Waterproof freezes on its first start using the dependencies, remove the dependencies using the tutorial in 8.2. Then reinstall but make sure Coq-SerAPI is selected in the 'Choose components' window. In addition, make sure to use the default installation location.

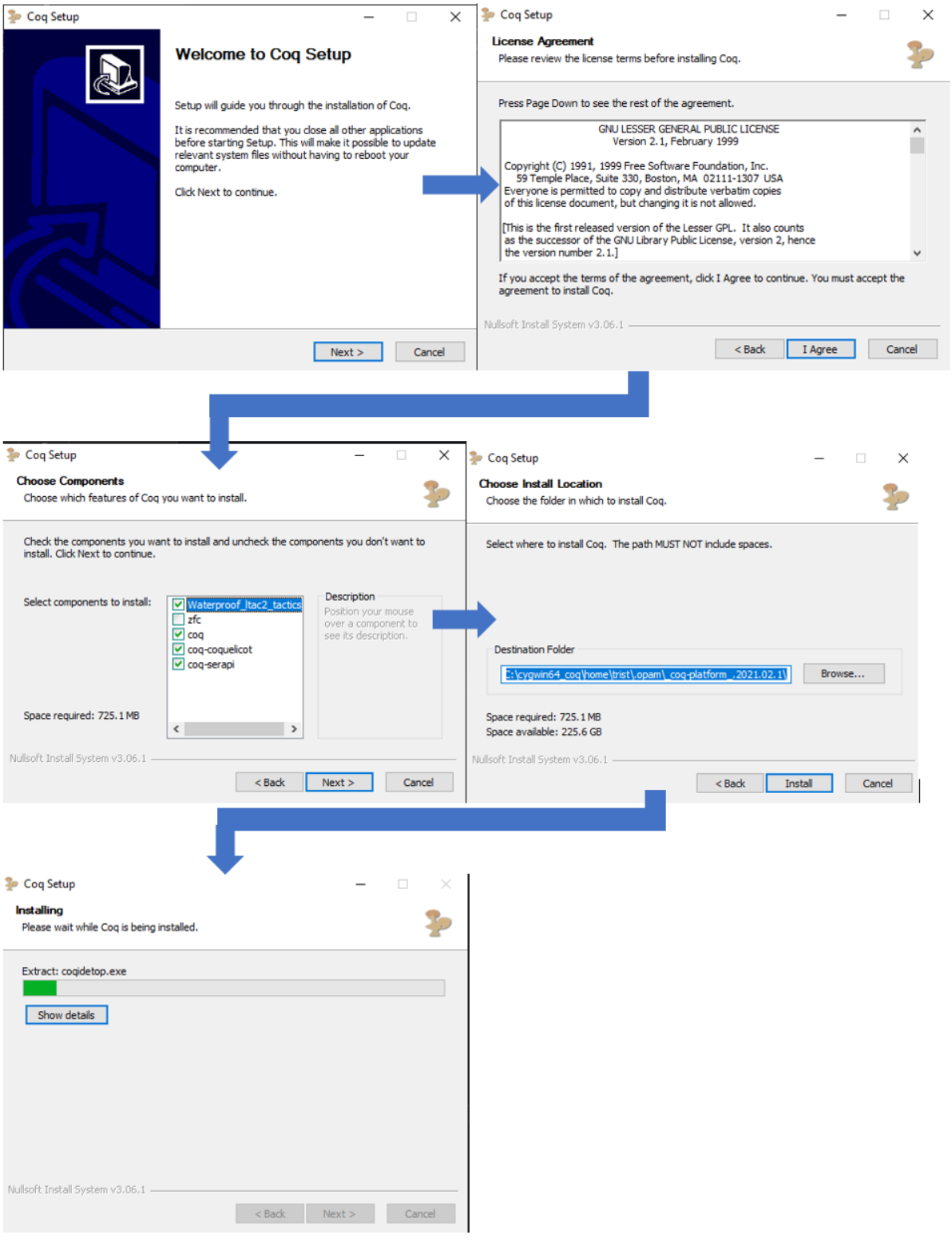


Figure 1: Installation process

8.2 Uninstall Waterproof dependencies

8.2.1 Functional description

This tutorial explains how to remove the Waterproof dependencies.

8.2.2 Cautions and warnings

The Waterproof dependencies must be installed and must not be running.

8.2.3 Procedures

1. Press the Windows key and search for 'Add or remove programs' system settings.
2. Open the settings.
3. Search for Coq Version X and click delete where X stands for the Coq version installed (8.13.2).
4. Press 'yes' in the pop up saying 'Do you allow this app from an unknown publisher to make changes to your account'.
5. The uninstaller window will pop up.
6. Click 'next' to start the uninstallation process.
7. Click 'next' to verify the location from which to uninstall.
8. The application will be uninstalled.

8.2.4 Possible errors

There are no documented possible errors for this tutorial.

8.3 Update Waterproof

8.3.1 Functional description

This tutorials describes how to update Waterproof.

8.3.2 Cautions and warnings

Waterproof must be installed and a new update should be available. And Waterproof should be closed.

8.3.3 Procedures

1. Open the Waterproof application.
2. A popup shown in Figure 2 will appear.
3. Click 'Yes, please' and close Waterproof.
4. The update will be applied when restarting Waterproof.

8.3.4 Possible errors

There are no documented possible errors for this tutorial.

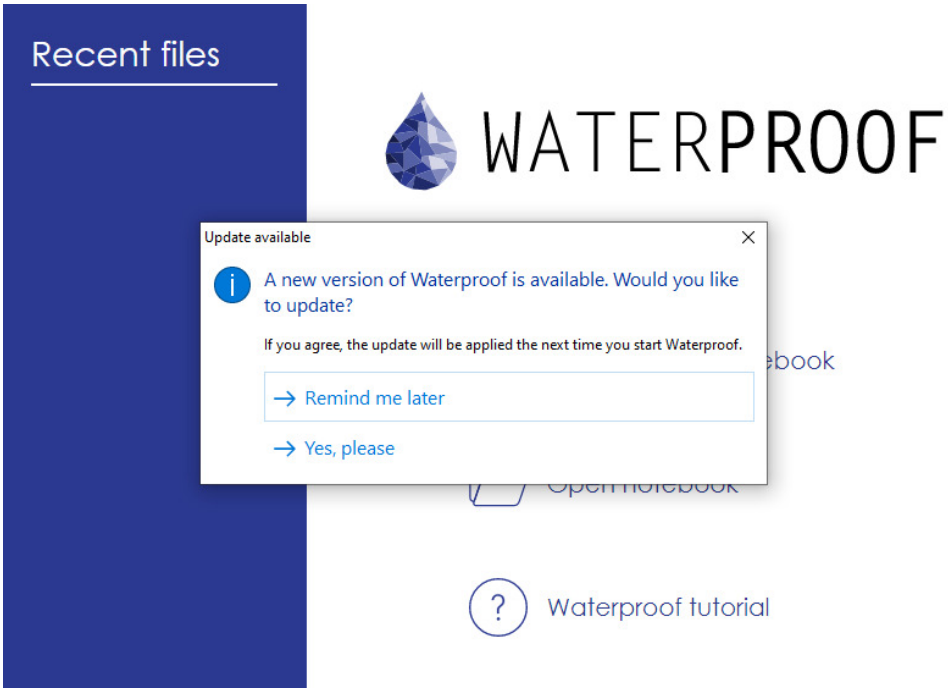


Figure 2: Updater popup

Part III

Abstract Syntax Tree

9 Introduction

9.1 Intended readership

This document is mainly geared towards **developers** of Waterproof familiar with the code. Parts of it, such as the Syntax highlighting tutorial, may be used by Waterproof users as well. However, most tutorials require readers to access files in the developer environment and knowledge of JavaScript.

9.2 How to use this part of the manual

This part of the manual will contain two main sections: tutorials and error messages with recovery procedures. No reference is needed since the tutorials cover the complete functionality.

9.3 Purpose

This section aims to teach developers how to make use of the current project and how to expand it further.

10 Overview

These tutorials are made to teach users how to use the syntax highlighting tool and, more importantly, teach developers how to expand the AST-parser and add new features in the future.

11 Tutorials

11.1 Syntax highlighting

11.1.1 Functional description

This tutorial explains how to use syntax highlighting.

11.1.2 Preconditions

- The user has opened a notebook in Waterproof.

11.1.3 Procedures

1. The user open the "Run" tab in the top left corner of the Waterproof program.
2. The user selects "Output all ASTs".
3. The syntax is highlighted according to a colour scheme generated by the names of the parsed types.

11.1.4 Likely errors

- The errors are the same as the errors for the pretty printer (cf. 11.3).

11.2 Extend AST-parser

11.2.1 Functional description

This tutorial explains how to expand the AST-parser by adding new types parsed from the CoqAST.

11.2.2 Preconditions

- Knowledge of the structure of the new Coq type being parsed.

11.2.3 Procedures

1. Open the folder `src/coq/serapi/datastructures`.
2. Create a new file with the name of the the new type being parsed from the `CoqAST`.
3. Include import statements at the top of the file for the following classes:
    - `convertToASTComp`
    - `CoqType`
    - `LocInfo`
    - `ASTVisitor`
4. Create a class with the same name as the filename which extends `CoqType` and implements `Visitable`.
5. Include a constructor in the class which takes as input an object of type `Array`.
6. Potentially include an `accept` method of type `void` which takes as parameter a variable of type `ASTVisitor`. If it is not implemented then the Visitor pattern will use the method implemented in `CoqType`.
7. Include a `pprint` method of type `string` which takes as input an integer parameter called `indent` with default value 0.
    - The parameter indicates the amount of indent when printing this type in occuring in a particular AST.
    - Create a constant variable as follows

      ```
      const tab = '\n'.concat('\t'.repeat(indent + 1));
      ```

    - Use this parameter at the end of any line written in the `pprint` method.
    - Note that you can use the `cprint` method from the parent class `CoqType` to print any output from `convertToASTComp`.
    - At the end of the method return `this.sprintf(super.pprint(indent), output)` where `output` is the string created for the pretty printer. This will wrap the content in the type name and parenthesis.

8. Write an export statement at the bottom of the file outside of the class as follows

```
export default ClassName;
```

9. In the visitor folder contained in the same directory, access the class `ASTVisitor`.
10. Include an import statement for the newly created class at the top of this file.
11. In the `ASTVisitor` interface include a method called `visitClassName` where `ClassName` is the same as the recently created class.
12. Include a parameter in this method called term of type `ClassName` or `CoqType` (depending on whether you chose to implement the `accept` method) and make the method of type `void`.
13. Include this `visitClassName` method in each of the features implemented through the visitor pattern. Notice that the method may remain empty without causing errors.
14. In the directory `src/coq/serapi` navigate to the file called ASTProcessor.
15. At the top of this file import the newly created class.
16. At the bottom of this file find the constant variable named `constrDict`.
17. Add an item to this list in the format

```
'ClassName': ClassName,
```

18. Return to the class you created called `ClassName`.
19. Add global variables for information contained in this Coq type.
20. Use the `LocInfo` class for location information.
21. Use the `converToASTComp` method for any remaining arrays that might contain another Coq type.

11.2.4 Likely errors

- Not applicable. The range of errors that a developer can introduce is too large to discuss.

11.2.5 Cautions and warnings

- Adding only partial implementations of a new type in the ASTVisitor may cause errors. For example, adding the newly created type in the interface list in the ASTVisitor file without implementing the method in each of the features implemented through the visitor pattern will cause the code to fail.
- Failing to add the new class into the `constrDict` variable will not give any errors; however, it will prevent the type from being parsed.

11.3 Pretty-printer

11.3.1 Functional description

This tutorial explains how to use the pretty-printer to output the ASTs of an open file.

11.3.2 Preconditions

- Have a non-empty Waterproof notebook open.

11.3.3 Procedures

1. If the console is not open, click `CTRL+SHIFT+I` to open it
2. Open the options in the run tab of Waterproof
3. Select the option "Output all ASTs"
4. Look at the output in the console log
5. The indent indicates the parent-child relationship. The name in parenthesis before an indent indicates the name of the type.

11.3.4 Likely errors

- The `VernacDefinition` type can throw an error in the constructor. This error is currently caught and ignored in the convert method and should not cause any problems.
- Some AST may contain unknown types. For these types, the raw array is printed without any structure.
- When using the pretty printer, the AST-flattening is also compiled. The visit methods for some types in the flattened visitor have not been implemented and can throw a "Method not implemented" error when the corresponding type appears. This can be fixed by commenting out the error throwing in the flattened visitor and/or implementing the flattened visitor for this type.

11.4 Visitor pattern

11.4.1 Functional description

This tutorial explains how to add new functionality over the AST by implementing a Visitor class.

11.4.2 Preconditions

- None

11.4.3 Procedures

1. Create a new class which implements the `ASTVisitor` interface.
2. Create the functions of the interface. All the functions follow the same functional signature

   `vist`*`TypeA`*`(term :` *`TypeA`*`) : void`

   where *`TypeA`* is the name of a class inheriting from `CoqType`.
3. Create the state variables which might be needed to store information. Bear in mind that the order in which terms of the AST are visited is dynamic. For example, the `FlattenVisitor` has a member variable state which contains all the pairs of Locinfo and keywords extracted so far.

4. Create a method that can be called to retrieve the current state from the visitor. In our code, we called this method `get()`.
5. (Optionally) To ease working with the newly-implemented Visitor, it is recommended to create a custom function which, given a CoqType, pass a new instance of the Visitor, and return the result. Assuming the new Visitor is called `NewVisitor`, the function would look like this:

```
function applyNewVisitor(ast: CoqType) {
    const visitor = new NewVisitor();
    ast.accept(visitor);
    return visitor.get();
}
```

11.4.4 Likely errors

- The errors in the visitor happen only if the developer himself introduces them. Thus nothing further can be said.

11.4.5 Cautions and warnings

- Not adding all of the types listed in the ASTVisitor interface to the new feature will cause errors that prevent Waterproof from running at all.
- For some parsed types, it is important to check if the global variables used for the feature are empty or not. For example, many of the variables in `VernacExtend` are often `null`. Failure to do so might lead to errors that are difficult to diagnose.

References

[1] Yves Bertot and Pierre Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. 2004. ISBN: 3-540-20854-2.

[2] Adrien Castella et al. *Waterfowl Software Design Document*. Eindhoven University of Technology, June 2021.

[3] Adrien Castella et al. *Waterfowl Software Transfer Document*. Eindhoven University of Technology, June 2021.

[4] Adrien Castella et al. *Waterfowl User Requirements Document*. Eindhoven University of Technology, June 2021.

[5] *Coq Package Index*. INRIA, CNRS and contributors. URL: https://coq.inria.fr/opam/www/ (visited on 06/22/2021).

[6] *Coq Reference Manual - identifier*. INRIA, CNRS and contributors. URL: https://coq.inria.fr/refman/language/core/basic.html#grammar-token-ident (visited on 06/29/2021).

[7] *Coq Reference Manual - term*. INRIA, CNRS and contributors. URL: https://coq.inria.fr/refman/language/core/basic.html#grammar-token-term (visited on 06/29/2021).

[8] Thierry Coquand, Gérard Huet, and Christine Paulin. *The Coq Proof Assistant*. Accessed: 2021-04-27. URL: https://coq.inria.fr/.

[9] *impermeable/waterproof*. original-date: 2019-09-20T07:42:09Z. Mar. 2021. URL: https://github.com/impermeable/waterproof (visited on 05/02/2021).

[10] *OCaml*. Accessed: 2021-04-27. URL: https://ocaml.org/index.html.

[11] *OCaml Package Manager*. OCamlPro. URL: https://opam.ocaml.org/ (visited on 06/26/2021).

[12] *OPAM archive for Coq*. INRIA, CNRS and contributors. URL: https://github.com/coq/opam-coq-archive (visited on 06/22/2021).

[13] Jim Portegies et al. *Waterproof Tactics Library*. URL: https://github.com/impermeable/coq-waterproof (visited on 06/29/2021).

[14] *The Coq Reference Manual*. INRIA, CNRS and contributors. URL: https://coq.inria.fr/refman/index.html (visited on 06/29/2021).

[15] *The Coq Reference Manual - Basic notions and conventions*. INRIA, CNRS and contributors, Feb. 2021. URL: https://coq.inria.fr/refman/language/core/basic.html (visited on 05/30/2021).

A Error messages and recovery procedures

A.1 Error messages for the tactics library

Error message:	`Unable to unify` *statement_of_constr* `with` *goal* (as in the Apply tactic).
Diagnosis:	Coq is not able to match the *goal* with the conclusion in *statement_of_constr*.
Recovery procedure:	Delete the line of code with this tactic and try another tactic, or another *statement_of_constr*.

Error message:	`AssumeError("Cannot assume premise: goal is not an implication")` (as in the Assume tactic).
Diagnosis:	The top-level connective of the goal is not an implication.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`AssumeError("Too many hypotheses provided")` (as in the Assume tactic).
Diagnosis:	Introduced more hypotheses than possible.
Recovery procedure:	Reduce the number of introduced hypotheses.

Error message:	`ChooseError("'Choose' can only be applied to 'exists' goals")` (as in the Choose tactic).
Diagnosis:	The top-level quantifier of the goal is not an ∃-quantifier.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`The variable` *constr* `was not found in the current environment.` (as in the Choose tactic).
Diagnosis:	The given *constr* does not refer to an existing variable.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`No such contradiction` (as in the Contradiction tactic).
Diagnosis:	Deriving a contradiction is not possible.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`The reference` *constr* `was not found in the current environment` (as in the Define tactic).
Diagnosis:	The given *constr* does not refer to an existing variable.
Recovery procedure:	Use an existing variable.

Error message:	`AutomationFailure ("Waterproof could not find a proof. If you believe the statement should hold, try making a smaller step.")` (as in the It holds that and We conclude that tactics).
Diagnosis:	The `waterprove` automation function was not able to automatically prove the respective claim.
Recovery procedure:	Delete the line of code with this tactic and try another tactic, or make a smaller step in the proof (with a statement that be proven in an easier way).

Error message:	`AutomationFailure("Could not verify that the proposition used for the rewrite holds. You may need to prove this proposition first before rewriting others with it.")` (as in the Rewrite inequality tactics).
Diagnosis:	Literal equality of proposition could not be verified by Waterproof.
Recovery procedure:	Delete the line of code with this tactic and try another tactic, or make a smaller step in the proof (with a statement that be proven in an easier way).

Error message:	`AutomationFailure("Given goal not equivalent to actual goal.")` (as in the We conclude that tactic).
Diagnosis:	The provided parameter is logically not equivalent to the top-most goal of the proof state.
Recovery procedure:	Provide a parameter that is logically equivalent to the top-most goal.

Error message:	`RewriteError("Could not rewrite goal with this expression")` (as in the Rewrite using tactics).
Diagnosis:	Given literal is valid, but cannot be applied to the current goal.
Recovery procedure:	Change the literal to something that can be applied to the current goal.

Error message:	`RewriteError("Rewriting the hypothesis with this equality is possible, but did not produce the expected result")` (as in the Write using tactics).
Diagnosis:	The result of the rewrite was different from the provided expected result.
Recovery procedure:	Provide the correct expected result.

Error message:	`RewriteError("")` (as in the Rewrite inequality tactic).
Diagnosis:	The chain of inequalities is not connected by transitive relations.
Recovery procedure:	Correctly write the chain of inequalities.

Error message:	`RewriteError("Invalid (in)equality connective")` (as in the Rewrite inequality tactic).
Diagnosis:	Syntactic mistake.
Recovery procedure:	Correct the syntactic mistake.

Error message:	`RewriteError("LHS of (in)equality must match LHS of goal.")` (as in the Rewrite inequality tactic).
Diagnosis:	A correct (in)equality cannot be applied to the current goal, but the left-most term in the 'chain' is not equal to the left-hand-side term of the goal.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`TakeError("The type of the variable must match the type of the 'forall' goal's bound variable.")` (as in the Take tactic).
Diagnosis:	The type of the introduced variable is different from the type of the bound variable of the ∀-quantifier.
Recovery procedure:	Use the correct type of the bound variable of the ∀-quantifier.

Error message:	`TakeError("'Take' can only be applied to 'forall' goals")` (as in the Take tactic).
Diagnosis:	The top-level quantifier of the goal is not a $\forall$-quantifier.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`No such assumption.` (as in the This follows by assumption tactics).
Diagnosis:	There is no assumption that is exactly equal to the current goal.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`GoalCheckError("No such goal")` (as in the Then holds by assumption and We need to show tactics).
Diagnosis:	The given parameter is not logically equivalent to the current goal.
Recovery procedure:	Replace the parameter with the actual goal.

Error message:	`ReflexivityError("Reflexivity cannot be applied here.")` (as in the This follows by reflexivity tactic).
Diagnosis:	The goal is not of the form $A = A$ for some mathematical object A.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`WeKnowError("This hypothesis does not exist.")` (as in the We know tactic).
Diagnosis:	The given hypothesis does not exist in the proof state.
Recovery procedure:	Replace the parameter with a hypothesis that does exist in the proof state.

Error message:	`No such assumption` (as in the We prove by induction on tactic).
Diagnosis:	Induction was applied to a variable that is not bound to a $\forall$-quantifier.
Recovery procedure:	Delete the line of code with this tactic and try another tactic, or use a variable that is bound to a $\forall$-quantifier.

Error message:	`BothDirectionsError("This is not an if and only if, so try another tactic.")` (as in the We show both directions tactic).
Diagnosis:	The goal is not an if and only if statement.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`BothStatementsError("This is not an 'and' statement, so try another tactic.")` (as in the We show both statements tactic).
Diagnosis:	The current goal is not a conjunction of two separate statements.
Recovery procedure:	Delete the line of code with this tactic and try another tactic.

Error message:	`BothStatementsError("None of these two statements are what you need to show.")` (as in the We show both statements tactic).
Diagnosis:	None of the given parameters are present in the conjunction that forms up the goal.
Recovery procedure:	Replace the parameters with the statements that actually make up the goal, as a conjunction.

Error message:	`RewriteError("Cannot rewrite the hypothesis with this term.")` (as in the Write as tactic).
Diagnosis:	The specified form is not by definition equivalent to the target hypothesis' original form.
Recovery procedure:	Replace the given form with the actual definition of the target hypothesis.

A.2 Error messages for the installer

Error message:	"Windows protected your PC".
Diagnosis:	Windows does not recognise the Waterproof dependency installer and blocks it from running, assuming it is a threat.
Recovery procedure:	Ignore the error and click 'More info' followed by 'Run anyway'.

A.3 Error messages for the AST

Error message:	"Method not implemented".
Diagnosis:	The ASTVisitor has attempted to use a visit method which was not implemented. The error will give a hierarchy of types. The last type to appear on this list corresponds to the method which was not implemented.
Recovery procedure:	Either comment out the throwing of the error in the method. This risks causing problems with the relevant feature. Otherwise, implement the respective visitor method. As no other feature has been implemented, the unimplemented methods will always be in the FlattenVisitor file.

Error message:	"Currently not parsing `TypeName`".
Diagnosis:	The type name will be given as the first index of the un-parsed array in the pretty printer as well. The structure of this new datatype can be analysed by printing the output of the basic parser to the console.
Recovery procedure:	Follow the procedure in 11.2 to parse the new type. To find out which keyword this type belongs to, follow the indices of the pretty printed ASTs to find which line the keyword is on.

Error message:	"Wrong argument provided to `ClassName`".
Diagnosis:	The `ClassName` is the type in which the error occurs. This error is due to the structure of the type in the CoqAST from SerAPI not matching the known structure implemented in the constructor.
Recovery procedure:	Analyse the output of the basic parser a modify the constructor to account for this variation of the structure as well. Another option is to stop parsing this type by removing it from the `constrDict` variable in the ASTProcessor file in the directory `src/coq/serapi`.

B Glossary

Term	Definition
API	Application Programming Interface
AST	Abstract Syntax Tree
Automation	A feature of Coq to automatically synthesize simple pieces of a proof.
Automatic solving	Coq commands that automatically advance the proof state.
Bash	Unix shell and command language.
Coq	A formal proof management system that allows for expressing mathematical expressions and assertions (cf. [8]).
CoqAST	A type of data structure returned by SerAPI which represents the code written by the user.
Coq-SerAPI	A library for machine-to-machine interaction with the Coq proof assistant.
Gallina code	The language used to write logical reasoning in a proof body, within the Coq system.
Git	A version control system, a tool used to manage and track changes of software projects.
GitHub	A provider of Internet hosting for software development, using Git.
Lemma	Proven statement used as a stepping stone for proving a larger result.
Mechanization	Automating a process for more efficiently accomplishing a given task.
OCaml	A general-purpose typed programming language. OCaml is designed to enhance expressiveness and safety (cf. [10]).
Opam	The OCaml Package Manager (cf. [11]). A source-based package manager for distributing OCaml programs and tools.
OS	Operating System
Proof assistants	"computer programs" specifically designed for mechanizing rigorous mathematical proofs on a computer.
Proof state	Dynamic "logbook" in Coq that shows (1) the current set of hypotheses and proven statements and (2) the statements that yet need to be proven.
SEP	Software Engineering Project
Software dependency	A software component necessary for the operations of a different program.
Software package	A collection of applications or code modules that work together to meet various goals and objectives.
Software repository	A storage location for software packages.
Syntax highlighting	The feature displays text, especially source code, in different colours and fonts according to the category of terms.
Tactic	Mathematical statement that advances the proof state.
UI	User Interface
URL	Uniform Resource Locator
User	Person that uses the application.

C Index

C.1 List of all tactics library tutorials

Tutorial	Purpose	Section
Configure the automation	The steps required to configure the automation of the tactics library	4.1
Tactics library installation through opam	The steps required to install the tactics library through opam.	4.2

C.2 List of all tactics

Tactic	Purpose	Section
Apply	Apply known result	5.1
Assume	Assume a premise	5.2
Because	Destruct hypothesis	5.3
Choose such that	Pick variable according to hypothesis	5.4
Choose	Choose specific variable	5.5
Contradiction tactics	Start proof by contradiction	5.6
Define	Define new variable	5.7
Either	Prove by case distinction	5.8
Help	Give hints	5.9
It holds that	Automatically prove a claim	5.10
Rewrite equality	Rewrite equality in goal	5.11
Rewrite inequality	Rewrite inequality	5.12
Simplify what we need to show	Simplify goal	5.14
This follows by assumption	Conclude proof using known assumption	5.16
This follows by reflexivity	Conclude goal by reflexivity	5.17
Unfold	Replace function by its definition	5.18
We claim that	Introduce sublemma	5.19
We conclude that	Conclude a goal	5.20
We know	Verify hypothesis exists	5.21
We prove by induction on	Start proof by induction	5.23
We show both directions	Start proof for implication	5.24
We shot both statements	Start proof for conjunction	5.25
Write as	Rewrite hypothesis	5.26

C.3 List of installer and updater tools

Tool	Purpose	Section
Dependency installer	Installs the dependencies required by Waterproof	8.1
Dependency uninstaller	Uninstalls the dependencies required by Waterproof	8.2
Waterproof updater	Updates the Waterproof software to the newest version	8.3

C.4 List of Abstract Syntax Tree tools

Tool	Purpose	Section
Syntax highlighting	Colouring syntax in Waterproof according to its meaning	11.1
Parser	Parses the CoqAST into JavaScript	11.2
Pretty printer	Prints the parsed object with the parent-child hierarchy	11.3
ASTVisitor	Interface for easily implementing new features	11.4