BACHELOR SOFTWARE ENGINEERING PROJECT

Software Design Document

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 Design Document

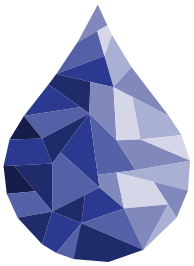
Team Waterfowl

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 0.6

July 1, 2021



Abstract

This document is the Software Design Document for the improvements made to the Waterproof software during the Waterfowl SEP project. Waterproof is a student-focused environment for writing mathematical proofs with a focus on real analysis. The design decisions presented in this document comply with the requirements stated in the User Requirements Project of the Waterfowl project. This document complies with the ESA standard (cf. [18]).

Contents

Document Status Sheet

Title	Software Design Document
Authors	Adrien Castella, Adrian Cucoş, Cosmin Manea, Noah van der Meer, Lulof Pirée, Mihail Ţifrea, Tristan Trouwen, Tudor Voicu, Adrian Vrămuleț, Yuqing Zeng
Version	0.6
Status	Final
Creation Date	May 4, 2021
Last Modified	July 1, 2021

Document History log

June 16^{th}, 2021: Initial unfinished release (version 0.1).

June 23^{rd}, 2021: Second unfinished release (version 0.2).

June 27^{th}, 2021: First draft release (version 0.3).

June 28^{th}, 2021: First revision based on supervisor feedback (version 0.4).

June 29^{th}, 2021: Second revision based on supervisor feedback (version 0.5).

July 1^{st}, 2021: Final revision (version 0.6).

1 Introduction

1.1 Purpose

This document contains the system overview (Chapter 2), system architecture (Chapter 3), and feasibility and resource estimates (Chapter 4) of the improvements made on the Waterproof software during the Waterfowl project. Waterproof is an interactive formal proof editor built on Coq, which provides feedback to the user regarding the correctness of mathematical proofs and features that enable easy access to common theorems that users may use within a particular proof.

This document aims to inform users (primarily developers) of the improvements made throughout the Waterfowl project: the creation of an installer and an updater for the back-end dependencies of Waterproof, the design of an Abstract Syntax Tree parser, internal to Waterproof and the development of a new tactics library for Coq.

Throughout the rest of this document, an in-depth description of the design of these improvements, the relationship with the old Waterproof software and its back-end dependencies, and the resources the software will now require are presented.

1.2 Scope

The Waterfowl project of improving Waterproof is being developed by a group of Computer Science students from the Eindhoven University of Technology as part of their Software Engineering Project. The project's goal is to improve the user experience of the mentioned application and facilitate updating and performing maintenance on it.

Currently, since Waterproof is built on top of the opam system (cf. [15]), maintenance and installation are complex tasks to perform. The tasks are to improve the user experience and maintainability of Waterproof.

In achieving this goal, an easy installer, whose purpose is to set up the underlying ecosystem and an updater, has to be created. Moreover, improvements to the Waterproof application have to be made. The most important ones are about the UI and its dependencies. To this end, syntax highlighting and a custom tactics library has to be implemented.

1.3 List of definitions

Term	Definition
Automation	A feature of Coq to automatically synthesize simple pieces of a proof.
Automatic solving	Coq commands that automatically advance the proof state.
Coq	A formal proof management system that allows for expressing mathematical expressions and assertions (cf. [9]).
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.
Cygwin	A program allowing the user to run native Linux applications on Windows.
Electron	A framework for creating cross-platform desktop applications using web technologies (JavaScript, HTML, and CSS).
Executable file	A program that can be run in the OS without explicitly needing to load it with another program.
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.
$L_{\text{tac}}1$	A meta language for Coq.
$L_{\text{tac}}2$	A newer meta language for Coq (successor of $L_{\text{tac}}1$).
Markdown	A format for text files that can be interpreted by a markdown viewer, but is also designed to be readable as source code. Typically uses the `.md` extension.
Mechanization	Automating a process for more efficiently accomplishing a given task.
Mutable variable	A variable in a program that can be changed in-place without needing to assign a new identifier to it.
OCaml	A general-purpose typed programming language. OCaml is designed to enhance expressiveness and safety (cf. [14]).
Opam	The OCaml Package Manager (cf. [15]). A source-based package manager for distributing OCaml programs and tools.
Portable executable	An application that embeds all its necessary dependencies and can be executed from a portable storage medium. Not related to the Windows `.pe` format.
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.
README	A markdown file commonly added to the root directory of a Git repository, that provided a summary of the content of the repository.

Serialization	The process of converting an object into a stream of bytes to transmit it or store it in memory, a database, or a file.
S-expression	A symbolic notation for representing a (nested) tree-structured data.
Sentence	Instructions used by Coq that can either refer to commands or tactics.
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.
Tactic language	A set of tactics that together form the complete functionality of Coq.
Transport Layer Security	A cryptographic protocol for secure communication over a computer network.
User	Person that uses the application.
Vernacular code	The language within the Coq system used to control the general behaviour of the program (e.g. for starting a proof, importing a module, printing a value, etc.).
`.v` files	Coq source code files (also known as vernacular files).
Wrapper	Application built around another program in order to provide a different interface. For example, Waterproof is a wrapper around Coq that simplifies the proving language.
`.wpn` / `.wpe` files	Files used by Waterproof. The extensions stand for "Waterproof notebook" and "Waterproof exercise sheet" respectively.

1.3.1 Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
INRIA	The French National Institute for Research in Digital Science and Technology
CPU	Central processing unit
JS	JavaScript
GB	Gigabytes
OS	Operating System
PDF	Portable Format Document
RAM	Random-access memory
SEP	Software Engineering Project
UI	User Interface
URL	Uniform Resource Locator

1.4 List of references

[1] *Agda*. Accessed: 27-06-2021. URL: https://github.com/agda/agda.

[2] *Auto Update - electron-builder*. URL: https://www.electron.build/auto-update (visited on 07/01/2021).
[3] 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.
[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 - Programmable proof search*. INRIA, CNRS and contributors. URL: https://coq.inria.fr/refman/proofs/automatic-tactics/auto.html (visited on 06/30/2021).
[7] *coq-serapi - Coq library*. June 2021. URL: https://github.com/ejgallego/coq-serapi (visited on 06/16/2021).
[8] *coq/platform*. Accessed on 2021-05-28. May 2021. URL: https://github.com/coq/platform.
[9] Thierry Coquand, Gérard Huet, and Christine Paulin. *The Coq Proof Assistant*. Accessed: 2021-04-27. URL: https://coq.inria.fr/.
[10] *Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS*. en-US. URL: https://www.electronjs.org/ (visited on 05/04/2021).
[11] *GNU make*. Free Software Foundation, Inc. URL: https://www.gnu.org/software/make/ (visited on 06/26/2021).
[12] *impermeable/waterproof*. original-date: 2019-09-20T07:42:09Z. Mar. 2021. URL: https://github.com/impermeable/waterproof (visited on 05/02/2021).
[13] *NSIS - Installer Framework*. June 2021. URL: https://nsis.sourceforge.io/Main_Page (visited on 06/14/2021).
[14] *OCaml*. Accessed: 2021-04-27. URL: https://ocaml.org/index.html.
[15] *OCaml Package Manager*. OCamlPro. URL: https://opam.ocaml.org/ (visited on 06/26/2021).
[16] *OPAM archive for Coq*. INRIA, CNRS and contributors. URL: https://github.com/coq/opam-coq-archive (visited on 06/22/2021).
[17] Jim Portegies et al. *Waterproof Tactics Library*. URL: https://github.com/impermeable/coq-waterproof (visited on 06/29/2021).
[18] ESA Board for Software Standardisation and Control (BSSC). *ESA Software Engineering Standards: Issue 2*. 1991.

1.5 Overview

The remainder of this document is divided into three chapters: the *System Overview*, the *System Architecture* and *Feasibility and resource estimates*. As it was previously described in Sections 1.1 and 1.2, the Waterfowl project is comprised of three major parts. These three parts are completely independent of a design point of view: creating an installer and an updater for the dependencies of Waterproof, the development of an Abstract Syntax Tree parser for the Waterproof GUI, and the development of a new tactics library. Each chapter in the remainder of this document will be split according to these components of the project.

Chapter 2 gives a general description of the improvements of the Waterproof application made during the Waterfowl project. This also outlines the relationship to other systems and the relationship to the previous Waterproof project. The re-implementation of the 'tactics library' is of particular interest concerning the old version of Waterproof.

Chapter 3 gives an explanation of the system architecture of the improvements made to the Waterproof application. This is a high-level overview of the design of modules, algorithms, interactions between objects, etc.

Lastly, Chapter 4 provides an overview of the resource use and demands that are needed to run the improved version of Waterproof (and each of its sub-components).

2 System overview

This chapter provides an overview of past, present, and future work related to the Waterfowl project. This overview includes the function and purpose of the Waterfowl project, the environment that the delivered software uses, and the project's relation to other systems.

2.1 Relation to current projects

The main goal of the Waterfowl project is to continue the development of the Waterproof GUI [12]. Waterproof is practically an advanced editor for the Coq Proof Assistant [9]. It encapsulates a Coq environment, allows a user to write non-interpreted text in between pars of a Coq script, and formats Coq's output into a more conventional mathematical notation. See Fig. 1 for a screenshot of the Waterproof GUI.

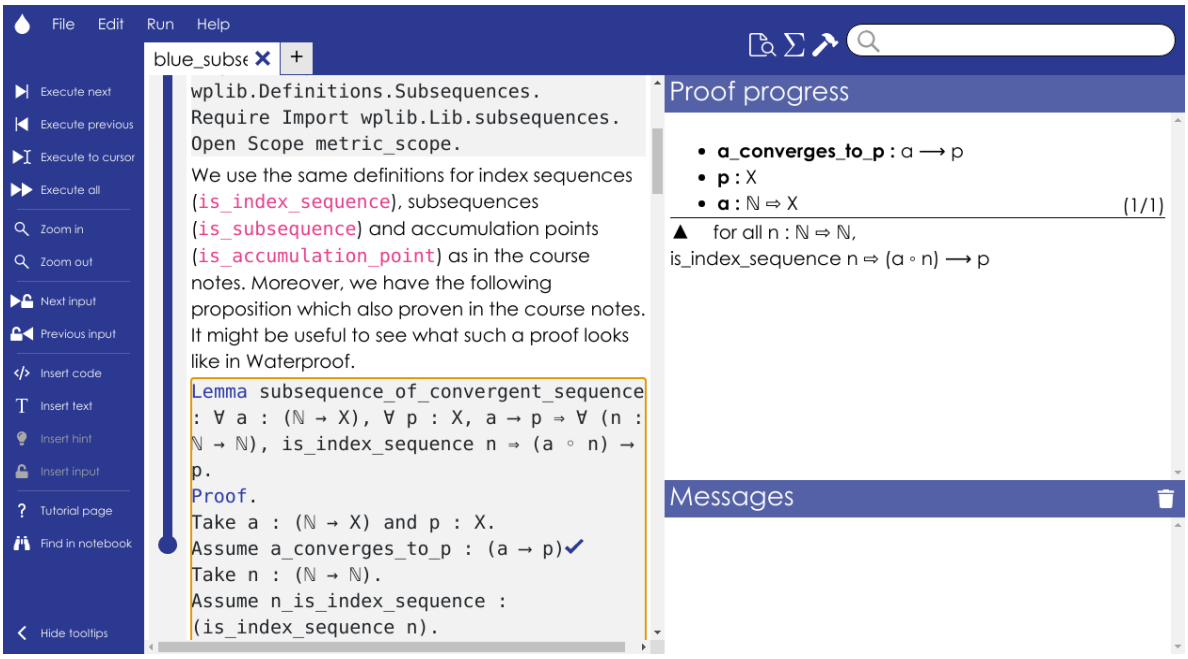


Figure 1: Screenshot of the Waterproof GUI with a 'notebook' opened. The middle pane shows the 'notebook', which is a Coq proof script interleaved with non-interpreted text. The right pane shows the reformatted output of Coq. The left pane are buttons for actions.

Waterfowl focuses on the following improvements for Waterproof:

- Creating a new installer for the set of dependencies of Waterproof. This does not change Waterproof's dependencies but makes it easier to install them for Windows users. The installer for the Waterproof GUI itself is not changed.
- A more advanced representation of the state of proofs within the Waterproof GUI. This feature is requested by the developers of Waterproof, who plan to use it to improve the user experience of the GUI.
- The tactics library used indirectly by Waterproof will be replaced. In particular, Waterfowl delivers a new tactics library written in $L_{\text{tac}}2$, while the old library was written in the older $L_{\text{tac}}1$ language. The tactics library is imported in the Coq environment that Waterproof encapsulates, and from there, directly used by the user of Waterproof.

2.2 Relation to predecessor and successor projects

This section will describe the systems that the Waterfowl project will either develop further or replace by a new system.

2.2.1 Waterproof Installer

A predecessor to our project is the Waterproof installer, which was developed prior to this project and provided the means to install Waterproof without any dependencies. This installer will be part of our project as is, where the installation process includes two installation files, one for the dependencies and the Waterproof installer, respectively. For a successful installation, the dependency installer developed by us needs to be executed before the Waterproof installer. These two installers together form the new installation procedure of Waterproof.

2.2.2 Abstract Syntax Tree parsing

In terms of parsing the data returned by SerAPI, the original Waterproof project implemented a basic parser that would convert an S-expression into an array of strings and objects from JavaScript. However, this parsing mechanism was relatively simplistic, and it was not used for processing the AST further. .

Waterfowl uses this basic parser as an intermediary format, from which a class-based AST parser is created, matching the internal structure of Coq. This way, none of the existing functionality in Waterproof, which relies on the basic parser, is broken.

In terms of successor projects, the AST-parser developed as part of Waterfowl can be augmented with support for more types. The current implementation has already some features which collect information about the currently unknown types and their occurrences inside a Waterproof Notebook. Additionally, more features can be implemented based on the AST-parser, such as creating an auto-completion database, using a similar procedure as the flatten operation.

2.2.3 The old tactics library

A particular predecessor of our project is the *old tactics library* (a concrete explanation of what a tactics library encompasses will be given in Section 2.5.6). This is a software package that is currently used by the Waterproof software. This project aims to develop a newer and more robust tactics library that replaces the old one. At the same time, this new library must offer *at least* the same functionality as the old library and must also be compatible with the rest of the Waterproof environment.

The new tactics library will then be viewed as a new dependency of the Waterproof software and, as such, will be integrated within the new Waterproof installer (cf. 2.2.1).

2.3 Function and purpose

This project adds specifically requested enhancements to the existing Waterproof software. These enhancements mainly target the back-end programs of the software package, although upgrades to the user interface are also made. The intent is to improve the maintainability of Waterproof from a developer perspective and also to improve user experience.

The enhancements can be grouped into three tasks, completely independent of each other concerning their design and structure:

- **Installer and Updater**: This task consists of making (1) a graphical installer for the back-end dependencies of Waterproof and (2) an updater for the Waterproof GUI (and the GUI's dependencies). The existing installer of Waterproof will be left unchanged since it does not require any modifications although it is part of the complete installation procedure.
- **AST parser**: The current Waterproof software requires users to input mathematical proofs in the form of vernacular Sentences. These commands are sent to SerAPI, which translates them to ASTs in an S-expression that Coq can interpret, known as a CoqAST. Coq processes this expression and then sends back a new AST containing its feedback for Waterproof. This AST feedback is sent back to Waterproof through SerAPI. However, it is not adequately parsed and remains in the form of an S-expression, which is not readable to the end-users. Only simple information is extracted from the returned S-expression, but no proper representation of the AST is performed. Hence, this project aims to include a parser that converts the CoqAST into a Waterproof-internal data structure to enable the implementation of new features that improve readability, such as syntax highlighting.
- **Tactics library**: the existing Waterproof software uses its library of custom tactics to simplify the Coq syntax. However, the old library depends on the outdated tactic language called $L_{\text{tac}}1$, which lacks some customisation options. The last task of this project is to create a new tactics library implemented using the more advanced tactics language $L_{\text{tac}}2$. The new library should offer at least the same tactics, and it also adds new features. The most significant improvements are that (1) tactics print more feedback for users, and (2) the library replaces Coq's built-in errors with more informative custom error messages.

2.4 Environment

Firstly, the task of making the installer requires that all dependencies and extra packages, such as the new tactics library developed by us, can be installed successfully using an installer which can be created and modified by the developers of Waterproof.

Secondly, the AST parser is made to parse specifically the CoqAST object already returned by Coq through SerAPI whenever code is compiled in Waterproof (for a description of these objects, see Chapter 3). The parsed information is made to help the developers make better use of the CoqAST, such as for syntax highlighting or automatic feedback. Therefore, the environment for the AST parser is the same as the environment required for Waterproof. The same applies to the syntax highlighting feature that is implemented alongside the parser.

Thirdly, the new tactics library is a stand-alone library for Coq (cf. [9]). It can be used with any operating system that can run Coq, including Windows 10 and Linux. Note that Coq depends on OCaml (cf. [14]) and opam, the package manager of the OCaml language (cf. [15]). The opam package manager is needed to download and install the library via the opam-coq-archive (cf. [16]), but is not needed at runtime.

Thus, we distinguish between two user roles, both of which shall operate using a machine running the Windows operating system, with at least 8GB of RAM and a CPU with at least 2 cores:

- **Waterproof user:** The Waterproof users are the end-users of the Waterproof interface. These users are assumed to be undergraduates Mathematics students, and use the software as a practice tool for courses in formal Mathematics.

Regarding the installer and the updater, these users can access the Waterproof GitHub page (cf. [12]) in order to download the necessary installation files for Waterproof. These installers can be run on the user's machine to perform a successful installation of Waterproof. Additionally, the Waterproof user can access the error logs generated by the installer application if any errors occur during installation.

Regarding the other two parts of the project, it is not expected that these users are familiar with Coq and the Abstract Syntax Tree parser, or even aware of the dependency of Waterproof on these two. As a consequence, they will also not be familiar with the specific syntax of build-in Coq tactics. Instead, they rely on the custom tactics library that comes with Waterproof, which provides a notation that is more akin to written mathematics.

- **Waterproof developer:** The Waterproof developers can access all modules that the Waterproof user role can. However, they have additional roles.

  Regarding the installer, the these users can utilise the Coq-platform (cf. [8]) and the additional scripts required in the creation and customisation of the installer. After creation, the installer can be distributed by the developer through the Waterproof GitHub page (cf. [12]).

  Regarding the Abstract Syntax Tree parser and the new tactics library, the evolution and maintenance of Waterproof will continue after the Waterfowl-project. For example, new features that depend on the AST may be added in the future or new tactics may be added or existing tactics might be modified. Therefore, it is important that the interfaces to the AST parser are clear and accessible to the developers and that the implementation of the tactics is modifiable.

The Waterproof relies heavily of SerAPI being present in the environment in order to communicate with Coq. Therefore SerAPI is required.

2.5 Relation to other systems

This section will explain the current software systems on which Waterfowl built its contributions.

2.5.1 Opam

Opam ([15]) is a package manager created for the OCaml programming language ([14]). This tool can be used to install and manage optional libraries for OCaml. It is used to install libraries for Coq, which makes opam relevant for the Waterfowl project. One of Waterfowl's installer requirements is that certain packages are downloaded and installed specifically through opam.

2.5.2 Coq Platform

The Coq Platform is a distribution of Coq which aims to provide a fast and robust way to install Coq and Coq plugins and libraries, as well as scripts to compile and install opam [8]. Furthermore, it is designed to be cross-platform, supporting Linux, Windows and macOS.

For Windows, the platform delivers a simple executable that installs a pre-compiled version of Coq and several packages, including Coquelicot.

Another instance of the aforementioned installer can be created with a shell script provided by the Coq Platform. This installer is created with the NSIS framework[13]. Furthermore,

the installer creation script also allows additional packages that the developer can choose to include as part of the installation prior to creating the installer. It is important to note that, although all opam packages can be installed, only installers that are created to include default packages (cf. Table 1) have been tested by the Coq Platform for proper execution.

Another file provided through the Coq Platform is a batch file that creates a Cygwin environment with opam and Coq pre-installed. It provides options for creating a full or a basic installation that automatically determines which opam packages will be installed. Opam packages can also be installed manually by opening the Cygwin environment since opam is installed there.

2.5.3 SerAPI

SerAPI [7] is a Coq package available through opam. Currently, SerAPI is still in an experimental stage of development and, while the project receives active support, no guarantees are made for long-term stability. Waterproof utilises the existing structure of SerAPI as means of communication between the front-end and the back-end, where Coq resides. Waterfowl also relies on the existence of the SerAPI package when creating an installer for Coq and other dependencies. Moreover, the AST parser depends on how SerAPI serialises messages to map Coq data structures to equivalent JavaScript objects.

2.5.4 Installer and updater frameworks

The installer of Waterproof dependencies relies on the installer framework used, namely NSIS [13]. Therefore, this framework is used to create this installer. A framework is a system that allows the creation of installation files, given a set of instructions specific to the framework used.

The updater of Waterproof relies on the Electron software.

2.5.5 Abstract Syntax Tree parser framework

Whenever a line of code is compiled in Waterproof, the code is sent through SerAPI to be compiled in Coq. Coq then returns its feedback through the so-called CoqAST as an S-expression. Next, it is received by Waterproof through SerAPI and then parsed into JavaScript through the basic parser which exists in Waterproof before this project. Finally, Waterfowl delivers a JS data type that encapsulates the output of this parser.

2.5.6 Tactics library framework

The new tactics library is technically a library for Coq [9], a program that Waterproof uses to check the proofs users write (see also Figure. 2).

Coq is an *interactive proof assistant*. This is a software package in which users can write mathematical proofs similar to writing in a programming language. More precisely, mathematical statements (such as lemmas or theorems) and their specific proofs can be written as Coq code for the proofs to be verified by a computer. Coq uses two languages to create proofs: Vernacular and Gallina (cf.[3]). Vernacular is used to denote the start and end of a proof, for printing output, etc. Gallina is the language used within the body of a proof and consists of so-called *tactics*.

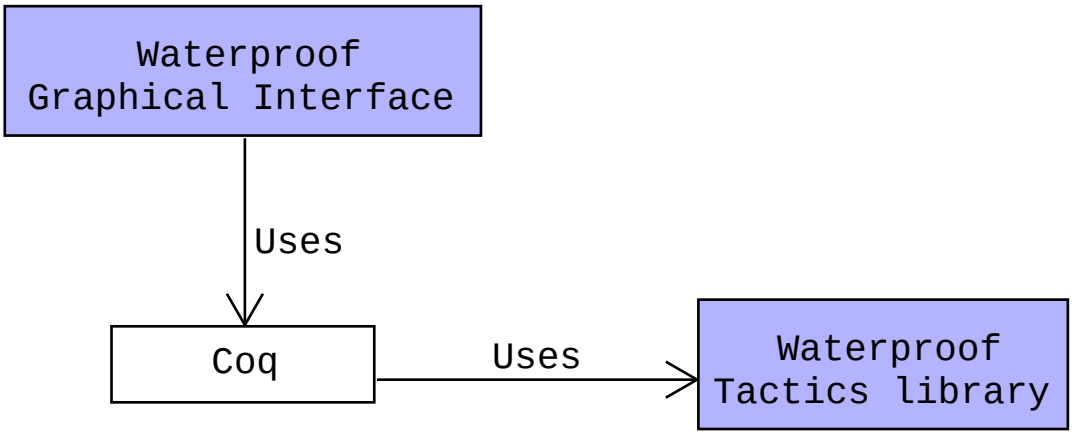


Figure 2: Indirect dependency of Waterproof's graphical interface on the tactics library: the library is technically a library for Coq. The blue elements highlight the modules that are part of the Waterproof project.

To assist the user in finishing a proof, Coq keeps track of the *proof state*. From the user's perspective, this is a dynamic logbook in which the current assumptions are shown and the statements that remain to be proven. *Tactics* are logical statements designed to advance the proof. Most tactics directly change the proof state. Examples of these are defining a variable, replacing a name by definition, applying a previously proven lemma, etc. Most tactics are an atomic operation, but a few tactics automatically attempt to advance or finish a proof.

It is possible to create user-defined tactics. This feature can, for example, be used to create macros that execute several built-in tactics. Custom tactics can also be used as a way to program proof-completing functions

Custom tactics are defined by using a *tactics language*. There exist several tactics languages for Gallina. Two relevant languages are (1) $L_{\text{tac}}1$, the first official tactics language of Gallina, and (2) its successor $L_{\text{tac}}2$.

The difference between $L_{\text{tac}}2$ and $L_{\text{tac}}1$ is that $L_{\text{tac}}2$ is more robust, supports more customizability, and is under more active development as $L_{\text{tac}}1$. The old Waterproof tactics library is implemented in $L_{\text{tac}}1$, and the new library primarily in $L_{\text{tac}}2$.

3 System architecture

3.1 Architectural design

This section will describe the modular structure of each of the three main parts of the project (cf. 2.3). Also, the concrete relationship between those three subprojects will be described. While each subproject implements a different feature in the back-end of the Waterproof graphical interface, they are not entirely independent of each other nor the Waterproof graphical interface. Fig. 3) shows a high-level overview of these dependencies.

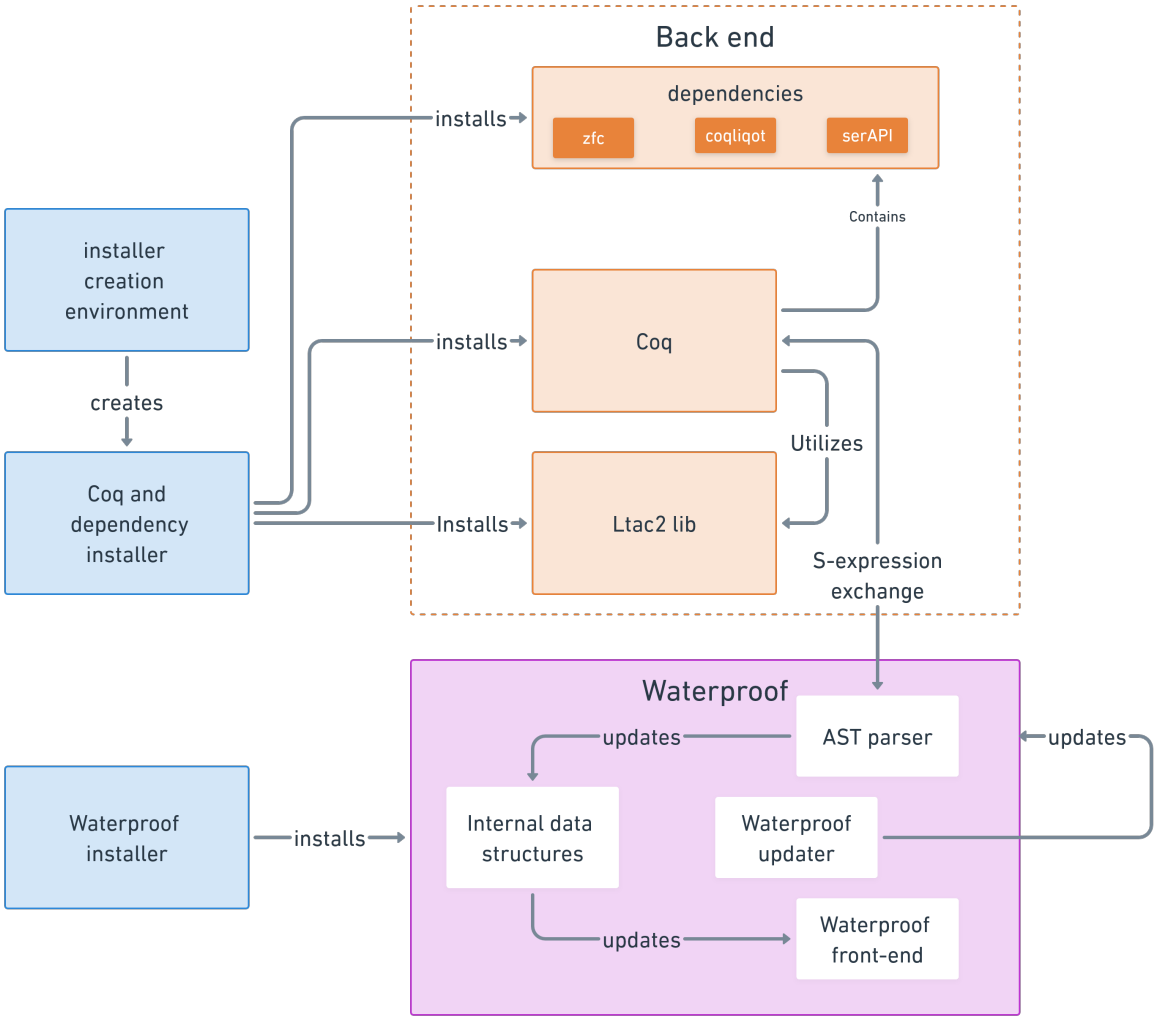


Figure 3: Flow chart indicating a high level overview of the relationship between the three different parts of the Waterfowl project. Colour coding distinguishes the setup files (blue) from the back-end (orange) and Waterproof (purple).

The installer creation environment should be used by the developer when creating a new installer for the Waterproof dependencies and back-end. The resulting dependency installer, together with the Waterproof installer, shall be run by the user to set up the Waterproof application.

The dependency installer installs Coq and other Waterproof dependencies such as coq-zfc, coq-coqliqot and coq-serapi as well as the new tactics library.

Waterproof is installed by the Waterproof installer and contains, among others: the AST parser, its internal data structures and the Waterproof updater.

Through direct communication between the AST parser and coq-serapi, Coq functions can be run directly from Waterproof and expressions written in Coq can be interpreted by Waterproof. Furthermore, the Waterproof updater is responsible for updating both Waterproof, and its dependencies.

The new tactics library (written in $L_{\text{tac}}2$) will be utilised by the Waterproof IDE front-end (which is practically an advanced IDE for Coq). Waterproof uses these tactics to provide a more accessible and more conventional notation for writing proofs than the one Coq uses by default. This allows the user to make use of Coq's functionalities without learning Coq at all. As previously mentioned, the new tactics library will be considered a dependency of Waterproof, and the new Waterproof installer will thus install it.

When a user writes tactics in Waterproof to create a mathematical proof, the AST parser processes and translates feedback received from SerAPI through a data structure called the `CoqAST` to communicate this feedback to users while allowing developers to make use of the feedback for other front-end features.

The following subsections will discuss the specific individual design of each of the three components.

3.1.1 Dependency Installer

The architectural design of the dependency installer is relatively short as it is a single executable file that was created using the NSIS framework [13]. The process of running the installer triggers a series of machine code commands executed to install the software.

3.1.2 Dependency installer creation environment

The developers of Waterproof use the dependency installer creation environment to customise and create an installer that, if successful, installs a complete set of dependencies required by the Waterproof application.

This section aims to explicitly differentiate between the systems used within the installer creation process while also showcasing the relationships between these systems. Consequently, one may refer to Figure 4.

The Coq platform [8] is downloaded from GitHub, which is responsible for installing Cygwin and Coq, as well as several other packages, which Coq can use. After performing these operations, the Coq Platform can build a dependency installer that can install Coq and several other packages. We will refer to this installer as the "user installer". It is to be noted that the Coq Platform is an entirely different system from the Coq back-end used by Waterproof.

The Install Manager system modifies the installer creation environment to accommodate more features, namely the inclusion of additional packages, which can be specified through the Installer Configuration file, and the customisation of the user installer created using the Coq Platform system.

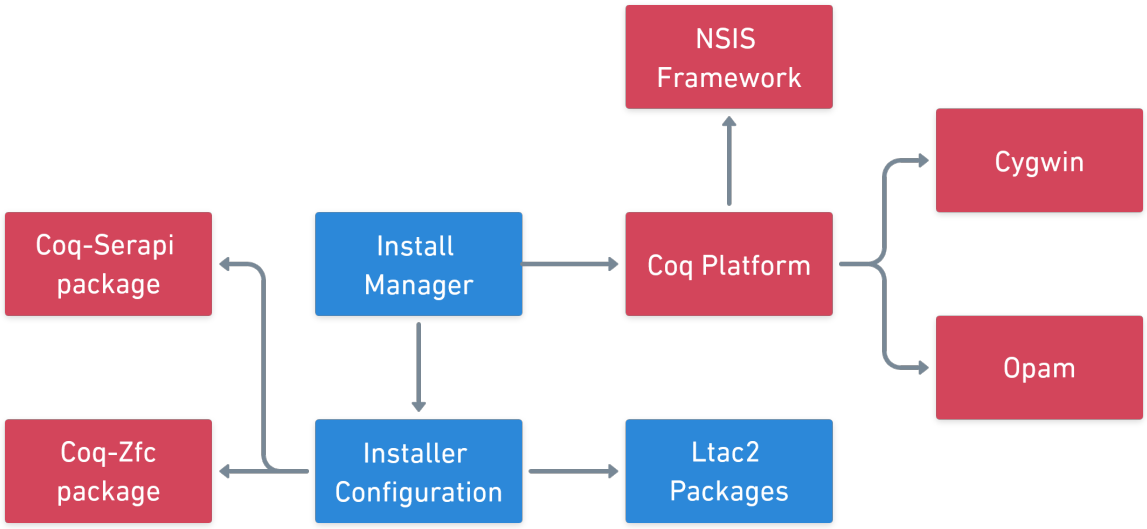


Figure 4: Flow chart indicating a high level overview that also distinguishes between the packages introduced by Waterfowl (blue) and other existing systems (red) interacting in the installer creation environment.

3.1.3 Updater

The Waterproof updater is based on the Electron AutoUpdater[2] component, which is included in the Electron framework used by Waterproof. For more information, refer to the documentation corresponding to the electron-updater package.

The updater architecture is based on the client-server model. The Waterproof application acts as a client interested in new software versions (i.e. updates), while the Github infrastructure fully handles the server-side. As the medium of communication between the client and the server, the internet is used. Note that the updater will only update the Waterproof software, and not say, back-end components such as Coq. An overview of the situation can be seen in Figure 5.

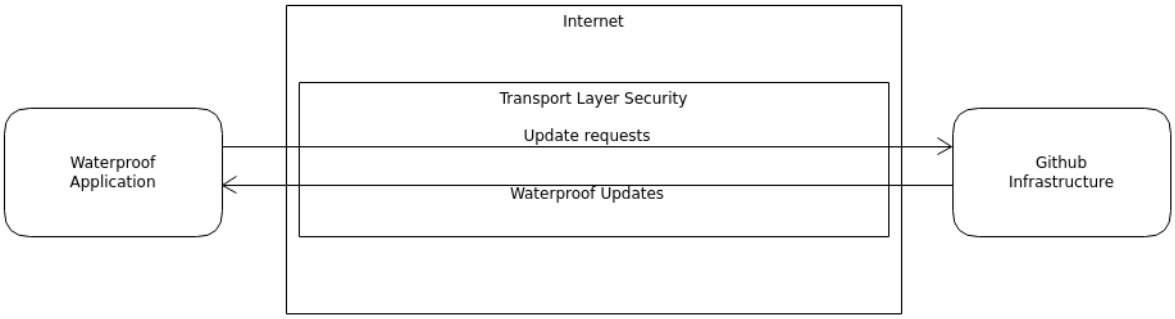


Figure 5: The updater architecture

In addition, the connection between the updater and the Github infrastructure is secured by Transport Layer Security, and all communication is performed over this secure channel. These are all mechanisms provided by the Electron Updater, which further uses the services provided by the operating system (e.g. DNS). Hence we will not go into further detail regarding this aspect.

3.1.4 Abstract Syntax Tree parser

The AST is an object returned by Coq through SerAPI in the form of an S-expression whenever any input, such as a tactic, is executed in Waterproof. The Waterproof editor sends the compiled code to SerAPI, which communicates with Coq, and then sends back an S-expression, which includes a data-type called the CoqAST. A CoqAST is returned for every line of code written in Waterproof. A line of code is defined as an expression ending in a period. Counter-intuitively, this can span multiple lines of text. This CoqAST data structure comprises many data types representing different tactics and Waterproof commands used in a notebook. For example, when proving a Lemma in Waterproof, the statement of the Lemma is represented by `VernacStartTheoremProof` data-type in the CoqAST for that line. On the other hand, the keyword to start a proof "Proof." is wrapped in a `VernacProof` data-type. Each of these data types has its own unique structure and can contain another nested data type, thereby turning the data structure into a tree.

The main concern of this part is to parse this CoqAST into a JavaScript object which preserves its structure. The parser will help future developers make better use of Coq's feedback whenever compiling a proof. The parser should include a feature to print the resulting JavaScript object in a way that preserves the parent-child relationship, called the pretty printer. Additionally, this object should also be used for syntax highlighting in the editor. In Figure 6, an overview of the interactions between the various components is given.

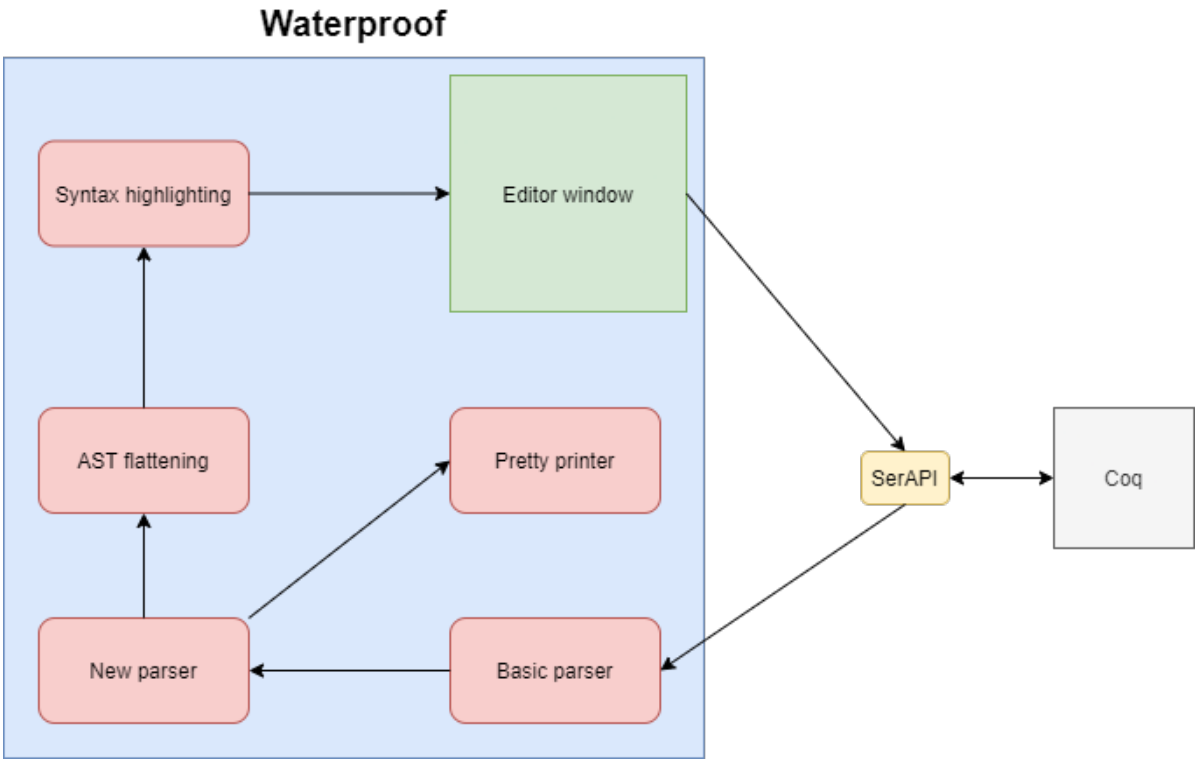


Figure 6: This system context diagram gives a high level overview of how the parsing should interact with the other components related to the AST.

As depicted in the figure, whenever an S-expression of type CoqAST is received from SerAPI, the information is processed by an already-implemented basic parser. This parser turns the S-expression into an array of strings and dictionaries, which the new parser can then use. This array is made to match the structure of the original CoqAST received from Coq,

which varies depending on the 90+ types present in the current tree. Therefore, it is not possible to give a detailed description of the structure of the array. Instead, the new parser implemented in this project takes this array output of the basic parser as input. It uses a pre-defined class structure for a subset of relevant types in the CoqAST to recursively parse the object into JavaScript. This structure then allows developers to make more flexible use of the information provided by the CoqAST. This can include applications such as providing additional proof feedback.

The pretty printer intends to allow developers to view the parsed information with its hierarchical structure preserved. It should use the class structure to determine the parent-child relationships.

The AST flattening is an operation that uses the parsed JavaScript object to create an array of pairs, including an index and a type. This should assign a type to each keyword in a proof based on the structure of the CoqAST. The index should naturally indicate the word's location, while the type indicates what the word is, for example, a Lemma. This data structure is meant to serve as a basis for syntax highlighting.

Finally, the syntax highlighting component, which uses the flattened array, assigns a colour to each keyword of the text written in the editor, which is then visualised when the proof code (the tactics and other Coq code) written by the user in the editor window is processed by Waterproof.

3.1.5 The tactics library

The $L_{\text{tac}}2$ implementation of the tactics library was developed as an independent Coq library and bundled as an installable package. This package is published as an open-source library to the opam-coq-archive repository (cf. [16] and [5]). As a result, Waterproof can install and access the library in the same way as other Opam packages.

The architecture within the library is straightforward. The tactics will be bundled as a modular collection of Coq files (with file extension `.v`). There is no class hierarchy since tactics are implemented as functions ($L_{\text{tac}}2$ is a functional programming language, which does not use classes or objects). Common routines are implemented as separate functions in "auxiliary" files to avoid redundancy in the tactics. Implementations of tactics can call functions from these auxiliary files. Most of the tactics are relatively simple functions that do not use extensive algorithms nor rely on other tactics. However, some of the tactics are more complex, as they rely on Coqś automation features (which in turn use so-called *hints databases*, described further below). These automation features, together with the database gathering, have been "wrapped" in a function called `waterprove`. To ensure maintainability and to avoid redundancy, the tactics only access automation via the `waterprove` function.

Most tactics are designed to throw an error and/or print a message when are incorrectly used (for instance, one cannot introduce a variable $n = 2$ when proving a statement of the form $\forall n, P(n)$ for some proposition $P(n)$). Some of the tactics also print messages containing recommendations for the user (for instance, proving by case distinction tells the user to specify each case they use concretely). This feature is added to enforce better proof writing, as explained in further detail in Section 3.6

3.1.5.1 Test cases $L_{\text{tac}}2$ does not provide a framework for unit tests. However, test-cases were needed, as the $L_{\text{tac}}2$ tactics library was published as an *official* opam package on the opam-coq-archive [16].

The testing was implemented in the following way:

1. The tactics library contains a collection of auxiliary testing functions. These functions compare an observed and an expected value, and raise an error if and only if the result is not as expected. There is, for example, a function that asserts a specific hypothesis exists, a function that compares the current goal with an expected expression, and even a function that expects an error to be raised.
2. Each source-code file has a test-code file counterpart. These test files consist of a series of tests that depend on the auxiliary testing functions. Since these functions raise an error if any test fails, the test files can only be compiled if all tests pass.
3. The `Makefile` (explained in Section 3.5.4.3) also compiles all test files, which enforces all tests to pass in order for the library to compile completely. From a practical point of view, this allows all tests to be run with a single command.

3.1.5.2 Hints databases. The more complex tactics use Coq's built-in automation features through the function `waterprove` (which is explained in more detail in Section 3.3.7). The behaviour of `waterprove` can be customised by supplying it with certain *hints databases*.

The term *hint database* is somewhat misleading, so it is necessary to clarify the semantics. A *hint* is a reference to previously proven theorems (and/or lemmas, which is just a synonym). The term *database* simply refers to a set of hints (and *not* to a collection of tables, such as SQL databases). Thus, one may think of *hints databases* as collections of theorems. Adding the right hints increases the number of problem types that the automation tactic is capable of solving.

One caveat is that theorems must be proven before adding them to a hints database. In Coq, it is impossible to refer to an unproven theorem, a feature that enforces correctness. However, it is possible to use theorems from existing libraries (in addition to proving new theorems).

A trade-off is involved here: adding more databases allows `waterprove` to finish more proofs but can also make it significantly slower. For this reason, the user can configure the set of databases used at runtime (as explained in Section 3.5.4.1).

3.2 Logical model description

In this section, a discussion of the logical model of each of the corresponding three parts of our project is given.

3.2.1 Dependency Installer Creation

For a visual overview of the files involved in in the installer creation see, Figure 7.

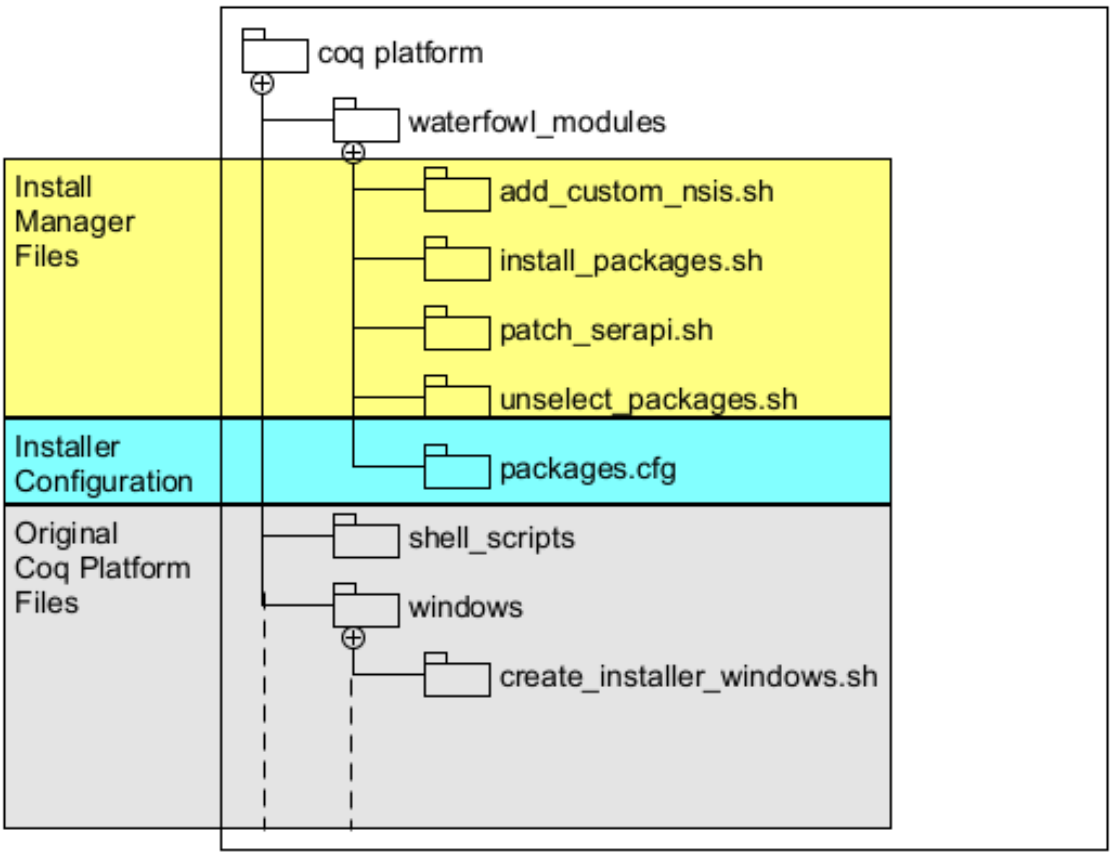


Figure 7: Indicates the files involved in the installer creation environment, after Coq Platform has been built.

Although there is no real communication between different scripts running inside the installer creation environment, it is the case that certain information is passed from one script to another through the creation of command lines in one script and their insertion into another. In some cases, the entire set of executable lines of code of one script is copied to another script. This is the case with

`unselect_packages.sh`, `install_packages.sh` and `add_custom_NSIS.sh`

scripts, which are copied inside

`create_installer_windows.sh`,

whenever the install_packages.sh is run. Furthermore, information such as package details can also be transferred from one script to another. Figure 8 is a component diagram

showcasing this process, where communicated information takes the form of an interface. It should be kept in mind that these are not interfaces in the classical sense but rather information with an exact structure, contained in one or multiple commands lines which ought to be copied.

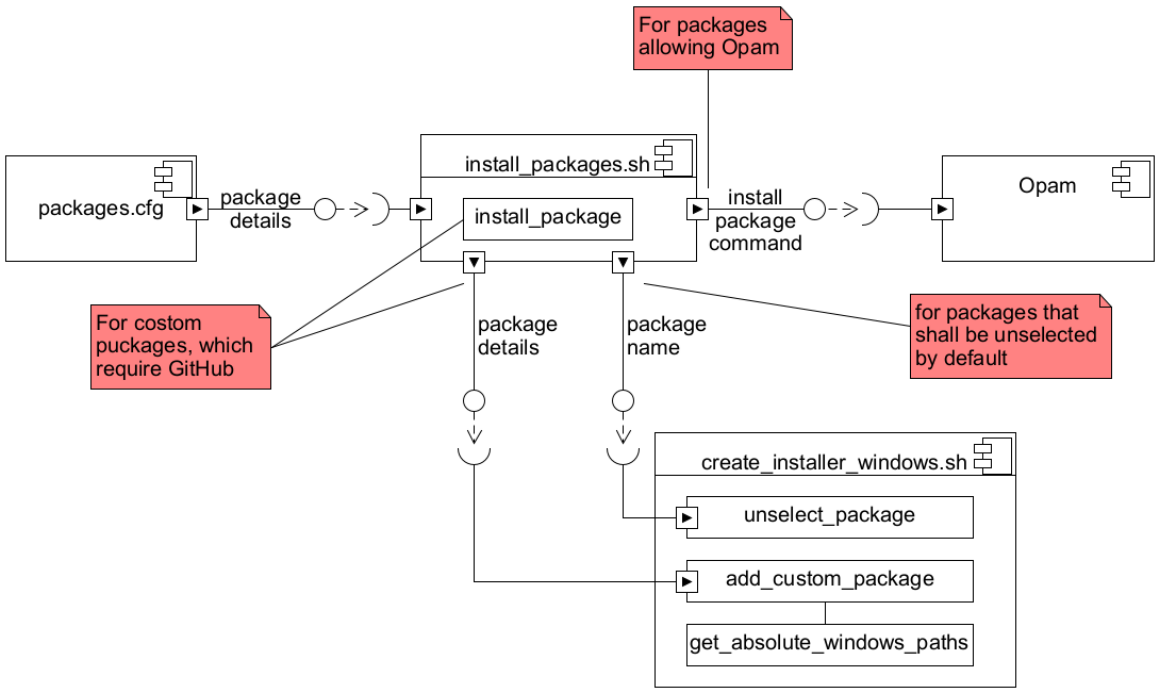


Figure 8: Component diagram representing the interaction between the scripts involved in the creation of the user installer.

Package details are first extracted from the `packages.cfg` file. The script

`install_packages.sh`

installs each package appropriately on the developer's machine. All the packages that require opam are installed using opam commands, and all the packages that need GitHub are installed utilising the function `install_package`. These packages are installed to have their contents converted into NSIS installer files. The script

`create_installer_windows.sh`

as part of the original Coq Platform can first convert the installed packages into NSIS installer files and finally create an executable installer. Initially, only opam packages could be used for creating an installer. This is why the script `create_installer_windows.sh` was changed as part of the Waterfowl project such that it can also support the creation of NSIS installer files from custom packages through the use of the function

`add_custom_package.sh`.

It is also significant that prior to execution of the `create_installer_windows.sh` script, the names of all packages which shall not be selected by default are communicated. This procedure ensures that all the packages are included in the resulted user installer and that their default selection status is set accordingly.

The exception to this process is the package Coq-SerAPI, which requires an additional step. After installation, the coq-serapi package requires the execution of script

```
patch_serapi.sh
```

, which is a used to repair the unusual behaviour of the Coq-SerAPI package. Although Coq-SerAPI is an opam package, it exerts unusual behaviour, making it difficult to include in the user installer. This is due to a path issue that does not allow the Coq-SerAPI package to be moved to another location after installation. Our team has informed the creator of this package about this issue and created this script as a temporary measure to this issue until its creator fixes the package.

3.2.2 Updater

The Waterproof updater is based on the Electron AutoUpdater [2] component, which is included in the Electron framework used by Waterproof. In addition, Electron popups are used for requesting user confirmation when an update is available. For more information, refer to the Electron documentation.

Figure 9 describes the relation between the Waterproof application and the Electron components during the updating procedure.

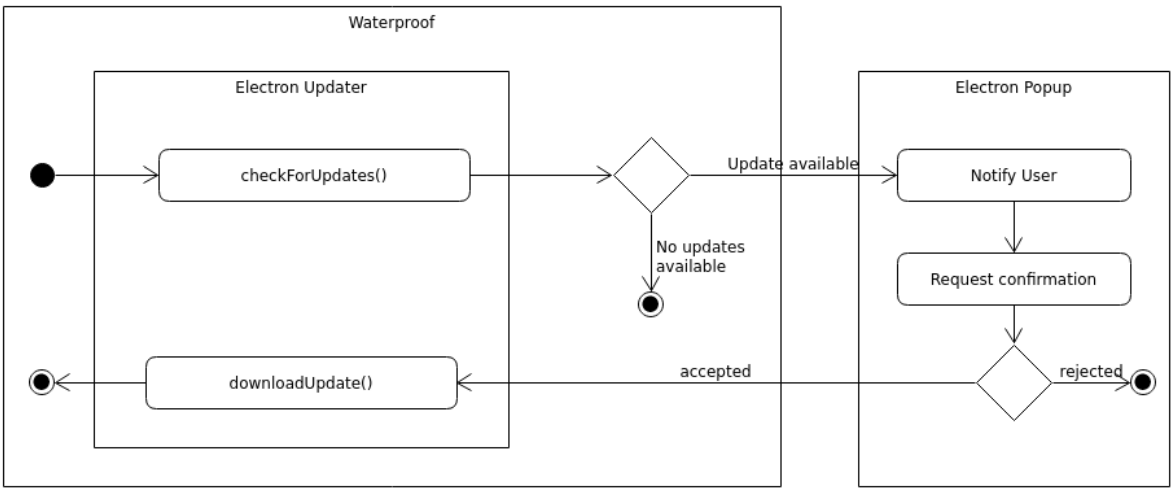


Figure 9: Diagram representing the relation between Waterproof and Electron components during the normal updater procedure

Note that the `checkForUpdates()` and `downloadUpdate()` functions are part of the Electron updater library . Refer to the corresponding documentation for more information (cf. [10]).

On the client-side, if the Waterproof application currently installed is several versions behind the latest release, updating the software will lead to the newest version being installed. Moreover, note that the updater will only update the Waterproof software and not say the back-end systems (which may widely vary) installed on user systems.

3.2.3 Abstract Syntax Trees

This section is dedicated to describing the logical model of the AST-parser. The data structure which needs to be parsed is called the CoqAST and is returned to Waterproof as an

S-expression by the intermediary, SerAPI. This data structure consists of many data types, each of which has a unique structure. Coq itself defines the data types. Note that, as mention in the architecture, the tree structure of the CoqAST results from the fact that each of these types may contain within it one or more other types. For example, some types, which are not explicitly defined, such as ones relating to tactics, appear within a `VernacExtend` type. Note that these types may not be explicitly defined in Coq, but they have a fixed structure that remains the same across any notebook in Waterproof.

To parse this data-structure, a JavaScript class is created for each relevant type from the CoqAST. As there is very little or no documentation for a large subset of the types, parsing must be done by analysing the output of the basic parser to understand the information relevant to the type. Therefore not all types are parsed, and out of the parsed types, there is no complex understanding for all of them.

Each class representing a type consists of a constructor which takes an Array as input and a pretty-printer method that prints the parsed content. Additionally, each pretty printer must print the name of the class or, equivalently, the type. Therefore, an abstract class called `CoqType` is defined, which is extended by each type.

The classes all implement a constructor which takes in the type-related information extracted from the input and inputs it into global class variables as JavaScript objects. These global variables and the parsing performed inside the constructor depend on the individual types which are being parsed. The pretty-printer uses the structure defined in the constructor to return a string representing the content. If any child types are present, it uses the respective pretty print method to perform recursion. To preserve the child-parent structure in the pretty printed output, the method takes as input an `indent` parameter, which is incremented for each child. This parameter indicates the amount of indentation used before each line of content in the resulting string. The inheritance of the types from the class `CoqType` is visualised in Figure 10 for three select types. As there are over 40 different parsed types, they cannot all be listed individually in this figure.

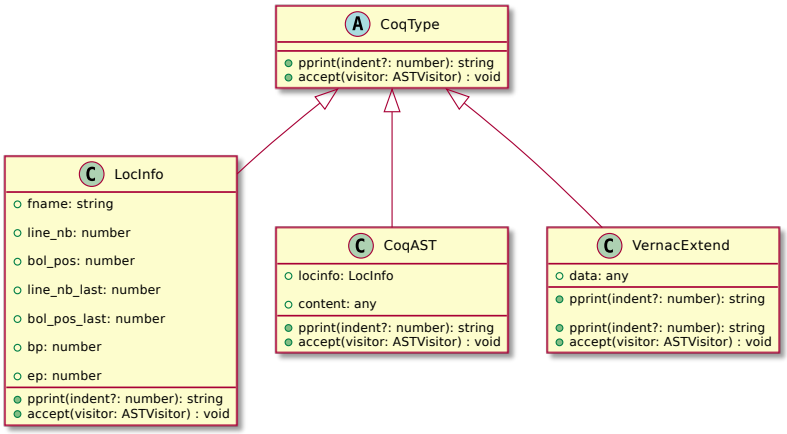


Figure 10: Illustration of the classes inheriting from CoqType.

The root type, which starts the recursion to parse the CoqAST is the `CoqAST` class. It is given in Figure 10. This class first stores the location information in `locinfo` given by the CoqAST using the `LocInfo` type and then start the recursion by using the `convertToAST` method. This method is defined outside of the type classes and is imported for each class known to have children. This method is used recursively to parse the entire data structure.

The basic parser, already implemented in Waterproof, outputs a JavaScript Array, as has already been mentioned. This array is then used as input for the `CoqAST` which then provides a subset of this array an input for the child types. Each parent, therefore, provides an Array input to the `convertToAST` method, which in turn provides this input to the corresponding child type. Therefore, the `convertToAST` methods takes as input an Array and then returns a new `CoqType` object. If the respective type has not been parsed yet (in other words, if no JavaScript class exists for it), then the method returns the original Array. A subset corresponding to some data type always stores the name of that type in the first element. Thus the method uses a dictionary called `constrDict` to determine whether a JavaScript class exists matching the string given in the first position of the Array. It then either returns a `CoqType` object which takes this Array as input or, if no JavaScript class exists, returns the Array and gives an exception indicating the name of the un-parsed type.

Aside from the `CoqAST`, the most important type is represented by the class `LocInfo`. As the title indicates, this type stores the location information of any type is it associated with. For instance, the location information of the CoqAST indicates the entire line of the proof. A child type such as `TacAtom` can have location information indicating specifically the start and endpoints of a tactic used in this line. The class structure can be seen in the previous Figure 10. The global variable `fname` describes the source of the input, either toplevel (independent) or the file name in which the sentence is contained. The type also stores the start position of the first and last line of the expression, which are most often identical. Finally it stores the start and end position `bp` and `ep` of the type respectively.

From here onwards, a list of all of the types that were parsed in this project and the structure of the data stored through the constructor. For the types for which it is known, a short description of their meaning is given. It is important to remember that each type in the CoqAST refers to a keyword used in the Waterproof code. Sometimes it can provide related information to this keyword, such as referencing some other variable used in the Waterproof code. For the meaning of tactics and other related subjects, please refer to the tactics sections of this document. Finally, it is crucial to notice that the location information provides that child's location whenever there is a pair of location information and a child type. Often this can be the exact location as the parent.

- The `VernacExtend` type extends the vernacular. It wraps any content for which Coq does not have an explicit type definition. The content is referred to as "generic arguments" and can be any of the types appearing in the CoqAST. It is stores in the global variable `data`.
- The `VernacRequire` type is used whenever the Waterproof expression "Require Import" is used. This type has a list of children, each given with separate location information. This list of pairs is stored in the global variable `list`. The types are `SerQualid` types, which indicate some ID or name of the import (for ex. Reals). `Qualid` consist mainly of an empty object.
- The `SerQualid` type consists of a Path variable `dirPath` which is mostly given by Coq as an empty object and an ID `id` which contains the name of a variable which this type concerns.
- The `GenericVType` type is simply a wrapper around any child type related to Vernac. Most often, we can skip the wrapper and parse the nested Vernac type directly. The child types are stored in `data`.
- The `VernacOpenCloseScope` type appears whenever a user closes or opens a scope in Waterproof. It contains two variables. A boolean variable `open` which indicates

whether it is opening or closing and a string variable `scope` giving the name of the scope.

- The `VernacStartTheoremProof` type appears at the start of any proof. The most important global variable is the `theoremKind` variable. This variable can take on various string values such as "Lemma", "Theorem", "Proposition", and more. It indicates the kind of statement that is being proven. The second variable `proofExpr` can contain other child types with location information pertaining to the proof. However, it is mostly empty.
- The `VernacEndProof` type appears whenever a proof is completed. It contains a string parameter `proofEnd` which expresses the end of the proof in natural language. In addition, it includes a `proofDetails` parameter, which indicates the style of the proof end (such as the indent used). Finally, the only important parameter is `proofFinished` which indicates whether or not the proof is in fact finished.
- The `VernacDefinition` type appears whenever a definition is given in Waterproof. The key information is in the `name` variable which contains location information and another nested child type. This type is an `IDt` type which, similarly to a `SerQualid` type, contains the name of the definition. The `VernacDefinition` type also contains a Boolean variable `discharge` indicating whether to "do" discharge or not. Finally, it contains a variable called `definitionObjectKind` which stores the type of the definition.
- The `IDt` type consists only of a name, which is a string variable. Appearing primarily in `VernacDefinition` it contains the name of a definition.
- The `VernacProof` type consists of two Array variables. It always appears between a proof start and the end of a proof. While the structure can be inferred from the output of the basic parser, its meaning is unknown. The type can still serve a use; however, knowing that it is associated with the keyword "Proof.", it is possible to assign a colour for the syntax highlighting.
- The `VernacExpr` type wraps around many types. However, it contains only one variable named `content` which has a single child type. It is mainly implemented to parse its children.
- The `VernacAssumption` type has two global variables. The content, stored in the global variables `discharge` and `inline` has no known meaning. However, a similar argument as in `VernacProof` applies. It can represent keywords such as "Hypothesis", which indicate an assumption.
- The `VernacHints` type contains a string and a type variable. The string, `strings` lists the names of libraries such as "Reals". The `hhintExpr` variable stores another child type relating to the hints. An example of such a type is the `HintsResolve` type.
- The `HintsResolve` type contains a list of types in the variable `hintList`. This can contain various types pertaining to hints such as the `HintsReference` type.
- The `HintsReference` type contains location information in `locinfo` and a reference contained in a `SerQualid` type in the variable `content`.
- The type `TacAlias` appears whenever certain tactics are used. Examples of such tactics are "Into" for introducing a variable, "specialise", or even "It holds that". The type has a location information in `locinfo` and a child type in `content` which is always a `KerName` type containing information on which tactics is used.

- The `KerName` is similar to the `SerQualid` type. However, rather than containing the name of a variable used in Waterproof, it is the name and ID of a tactic. These are stored in `id` and `type` respectively.
- Some other tactics are instead contained in the `TacAtom` type. This can, for example, be a "Rewrite" tactic. The `TacAtom` type refers to atomic operations such as the apply tactic in `content` together with location information in `locinfo`. The tactic is given as a child type and may have varying structures.
- The apply tactic has its own type, namely `TacApply` which contains location information in `locinfo` and a reference to some variable in `content`. This global variable consists of a child type, which can be a `CRef` explicitly referencing the variable.
- A `CRef` type contains location information in `locinfo` and the variable that it is referencing in `content`. This global variable often consists of a `SerQualid` type mentioned before.
- A `TacticDefinition` type is used to indicate the definition of a new tactic notation. This applies to both $L_{tac}1$ and $L_{tac}2$ tactics. It has a variable `type`, which indicates what type of expression it is, for example, an $L_{tac}1$ expression. Furthermore, it contains location information in `locinfo`. Finally, in the `content` variable, it contains other children types to specify the tactics or Coq code used to define the new tactic notation.
- The `TacFun` type contains both a name and a child type. The variable `name` is the name of some variable defined in a Waterproof notebook, while the `content` can be some child type referring to a tactic such as `TacThen`.
- The `TacThen` can have multiple children stores in the Array variable `content`. The possible types of children are the Tac data-types discussed in this section. For example, it can include another `TacThen`.
- The `TacAtom` type has a location info variable `locinfo` and variable `content` which contains a child. The child can be a variety of types.
- The `TacReduce` type has a variable `type`, which is a string variable that takes on values such as 'Unfold' (the tactic). It also contains location information in the variable `locinfo` and a child type stored in `content`. This type is often a `SerQualid` type making reference to some variable name with which the tactic is used.
- The `TacRewrite` type contains location information in `locinfo` and a child type in `content`. This type is a `CRef` referencing the variable being rewritten with the rewrite tactic. The variable name itself is stored in a `SerQualid` type inside the `CRef`.
- The `TacCall` type contains location information in `locinfo` and a child type in `content`. The stored type is always a `SerQualid` type, which can reference multiple things such as the type (ex. $L_{tac}1$) or a tactic itself (ex. reflexivity). It can also contain a reference to some previously defined variable in Waterproof. This is contained in `reference` which is an Array containing location information and another child, which is a `SerQualid` type with the name.
- The `TacArg` type contains location information in `locinfo` and a child type in `content`. Similarly to `TacThen` this child type can be another other Tac type.
- The `TacIntroPattern` type has location information in `locinfo` and a child type in `content`. This child is of type `IntroNaming` and contains the name or Id of a related variable.

- The `IntroNaming` type contains only a `content` variable consisting of a single child, which is typically an `IntroIdentifier`.
- The `IntroIdentifier` type contains only one variable, namely `Id` which is a string variable indicating the name of a variable in Waterproof.
- The `CApp` type contains a `list`, which is an Array of pairs of location information and child types. It also includes a variable called `first` which contains the project flag (a simple string) and another pair of location information, and a child type.
- The `CLambdaN` type contains a variable called `localExprs` which can contain a list of nested CoqASTs. Notice this is not a simple child as children are not typical of the CoqAST type. The constructor retrieves the CoqAST through the `extractCoqAST` method, which searches for the keyword "CoqAST". The second variable is called `expr` and contains a pair of location information and another standard type.
- The `CLocalAssum` type contains a list of pairs of location information and children in the variable `names`. It also includes a string in `binderKind` indicating its type. Finally, it contains another key pair of location information and child in the variable `expr`.
- The `CNotation` type contains a child in the variable `notation` indicating the notation used in the relevant line. Additionally, the dictionary `constrNotationSubstitution` contains three strings and a list of pairs of location information and other children.
- The `CPrim` type contains a boolean variable `isNumeric` indicating whether the referenced text is numeric or not. Additionally, it contains a variable called `value` which can either contain a string or a dictionary indicating the positivity of the number and some other details.
- The `CProdN` type consists of multiple children and a single location information. The variable `localExprs` contains the list of children, while the variable `expr` contains the location information and another key child to which the location information belongs.
- The `InConstrEntry` type simply contains an Array named `data`. The content of this Array is not well known. However, as mentioned before, it can still serve a purpose for certain features.

Notice that all of these classes we have mentioned inherit from `CoqType` and were made to imitate the structure of an existing type of the original CoqAST data structure.

Using these pre-defined class structures for parsing the CoqAST provides the opportunity to define an interface for creating new features based on the CoqAST, such as the previously mentioned flattening used in Syntax highlighting. It still remains flexible as new types can easily be implemented without disrupting the code. To allow for this flexibility, the Visitor pattern is for this interface.

The visitor pattern is visualised in Figure 11. We use a single arbitrary CoqType class to visualise it for the same reasons as in the previous Figure. The `ASTVisitor` is an interface. Any class which implements this interface is required to implement each of the abstract methods listed in the `ASTVisitor` interface. Moreover, each class inheriting from `CoqType` must implement the `accept` method which takes as argument an `ASTVisitor`.

This allows for a special form of communication called double-dispatch, in which a Visitor object can call the accept method of a `CoqType`, which in turn, dynamically calls the appropriate forms on the visitor.

By implementing this design pattern, the functionality we want to implement by traversing the AST generated from the sentence is decoupled from the actual implementation of the `CoqTypes`.

Consider, for example, that we want to implement a new feature using the AST-parser, which gathers statistics about the classes occurring in a Coq sentence. The only thing we have to do is create a new class inheriting from the `ASTVisitor`, and overwrite each of the methods presented by the interface. Thus, we do not have to alter any of the classes inheriting `CoqType`. Subsequently, if we add support for a new `CoqType`, say `TypeB`, the other classes inheriting from CoqType remain unchanged, and we only need to add one method to the `ASTVisitor` interface, `visitTypeB`, and implement it in the classes inheriting `ASTVisitor`.

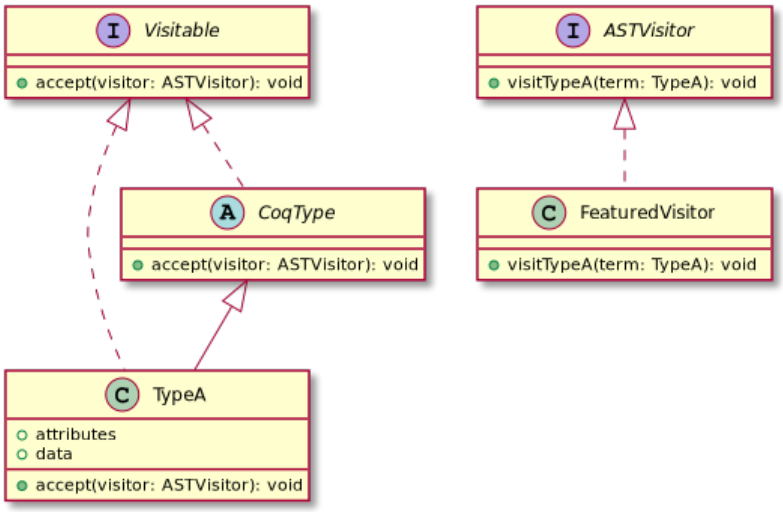


Figure 11: Illustration of Pretty-Printing with Visitor Pattern.

3.2.4 Tactics library

Let us now discuss the logical design of the new $L_{\text{tac}}2$ tactics library. This discussion is again divided into two parts: (1) the "stand-alone" tactics and (2) automated tactics that use hints databases.

3.2.4.1 Tactics Every tactic is implemented as a function. Shared functionality between tactics is implemented as auxiliary functions in separate files. $L_{\text{tac}}2$ is a functional programming language, implying that the library does not contain classes or objects. Except for the *rewrite-inequalities chains*, the only data structures used are simple lists of constants and lists of tuples.

Chains of inequalities The tactic `Rewrite inequalities` takes as input a 'chain' of inequalities of the form $e_1\ R_1\ e_2\ R_2\ \ldots R_{n-1}\ e_n$, where each $e_i \in \mathbb{R}$ and each $R_i \in \{=, <, \leq, >, \geq\}$. Before initiating the rewriting algorithm, this chain in mapped to a simple data-structure. This data-structure is an Ltac2 list of `Chain` terms, where `Chain` is a custom Ltac2 type. There are two ways to make a term (literal expression) of type `Chain`:

- The literal term `ChainEnd`.
- The constructor `ChainLink`. This constructor takes as argument a tuple of the form (e_i, R_i, e_{i+1}) (again with $e_i, e_{i+1} \in \mathbb{R}$ and $R_i \in \{=, <, \leq, >, \geq\}$). In Ltac2, a constructor

'wraps' an expression such that it has the desired type (`ChainLink`). Terms of this type represent an individual (in)equality in the chain.

For each (in)equality in the input, a `ChainLink` is made, and included in this list. The last item in the list will be a `ChainEnd`. As a consequence of encoding each individual (in)equality explicitly[1], the expression e_i appears twice in the chain for $i = 2, 3, \ldots, n-1$ (once as the left-hand-side of an (in)equality, and once as the right-hand-side).

3.2.4.2 Hints databases

One significant component of the tactics library is the inclusion of *hint databases*.

Recall that a hint is a lemma (a proven proposition). For example, *for all* $x, x + 0 = x$). A collection of hints is called a hints database. One can refer to a hints database by a name, similarly to a variable. The tactics library specifies all the used databases (including the lemmas they contains) in a single `.v` file. Additional lemmas and/or databases can be easily added by extending this file..

Automation The Waterproof tactics library uses the hints databases in its automated tactics. All the automatic tactics make use of a function called `waterprove`, which can be used to attempt which can be used to attempt to solve a goal using the specified databases automatically. The running time of `waterprove` depends on the amount of databases loaded, therefore it is unpractical to always include all available databases by default. Furthermore, the running time of `waterprove` also depends on the *search depth*. The `search depth` can be used to specify how many reasoning steps `waterprove` can apply before trying a different approach [6]. Note that not all reasoning steps may have a 'depth cost' of 1. Section 3.6.4.6 explains why automated tactics and hint databases are needed. How the databases can be configured is explained Section 3.5.4.1, and the running time of `waterprove` is further explained in Section 4.4.1.

Automation state model The tactics library specifies two global variables,

`global_database_selection`

and

`global_search_depth`.

The variable `global_database_selection` is a list of enumeration-instances that represent the collection of selected hints databases. When `waterprove` is called, it will read this list and map the enumeration instances to the actual databases (in $L_{\text{tac}}2$, it is not possible to create a list of databases references directly).

And, as its name suggests, the variable `global_search_depth` is simply an integer that encodes the search depth used by `waterprove`.

[1] This is intuitive in an example. Consider the proposition $p := a < b < c$ for some $a, b, c \in \mathbb{R}$. Note that p is practically a conjunction of two hypotheses $p_1 := a < b$ and $p_2 := b < c$. Note that b appears *both* in p_1 and in p_2, but only once in p.

3.3 State dynamics

3.3.1 Installer creation

In order to automate the process of creating an installer, a GitHub Workflow has been set up for this purpose. The developer will have access to the created installer through GitHub after this process is done. A workflow file has a `.yaml` extension and describes the scripts together with the parameters which they ought to be run with to produce the installer. The entire process is showcased by the sequence diagram in Figure 12. First, the developer shall edit the configuration file to include all the packages that can be installed through the created installer, together with their default selection status. Then the developer runs the GitHub action responsible for creating the installer. Finally, the developer can download the finished installer. This procedure concerns the A.9 and A.10 use cases of the URD [4].

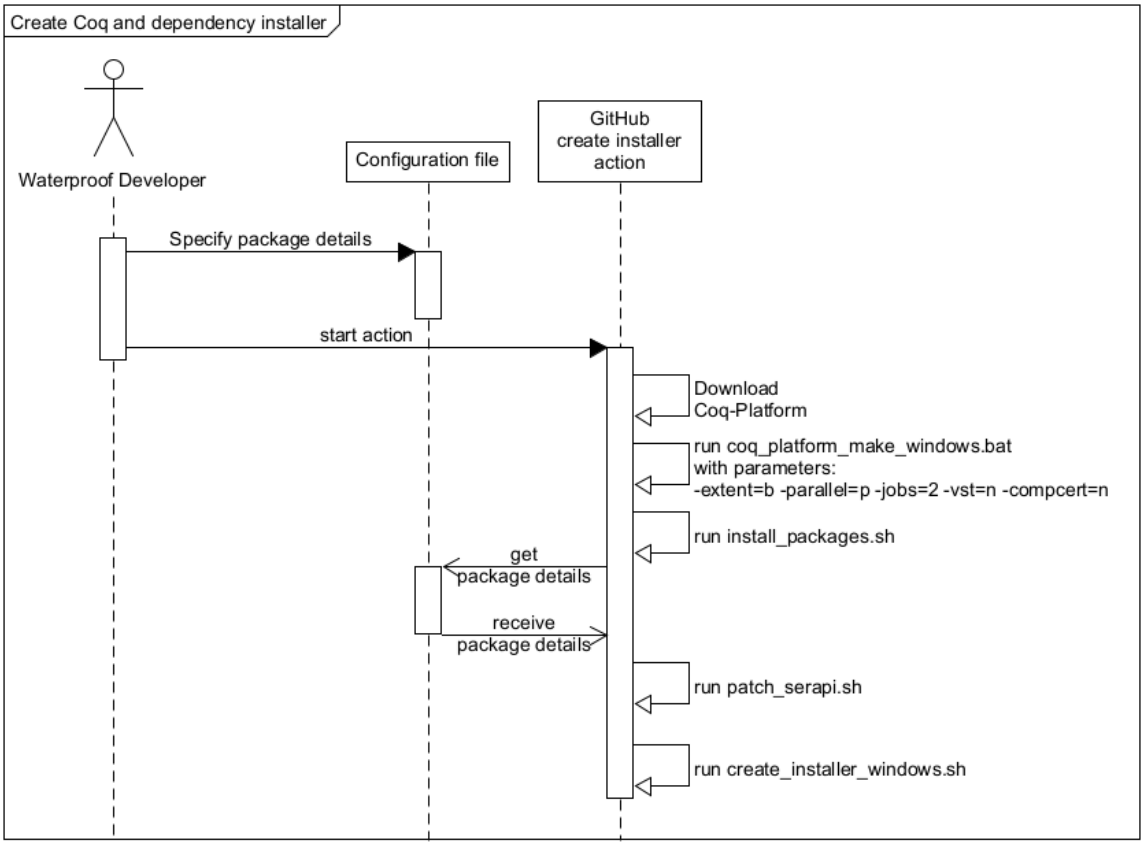


Figure 12: Creation of Coq and dependencies installer.

3.3.2 Dependency Installer usage

Figure 13 shows a sequence diagram for the dependency installer execution process. To install Waterproof dependencies, the user first runs the installer. The installer responds by displaying the user interface.

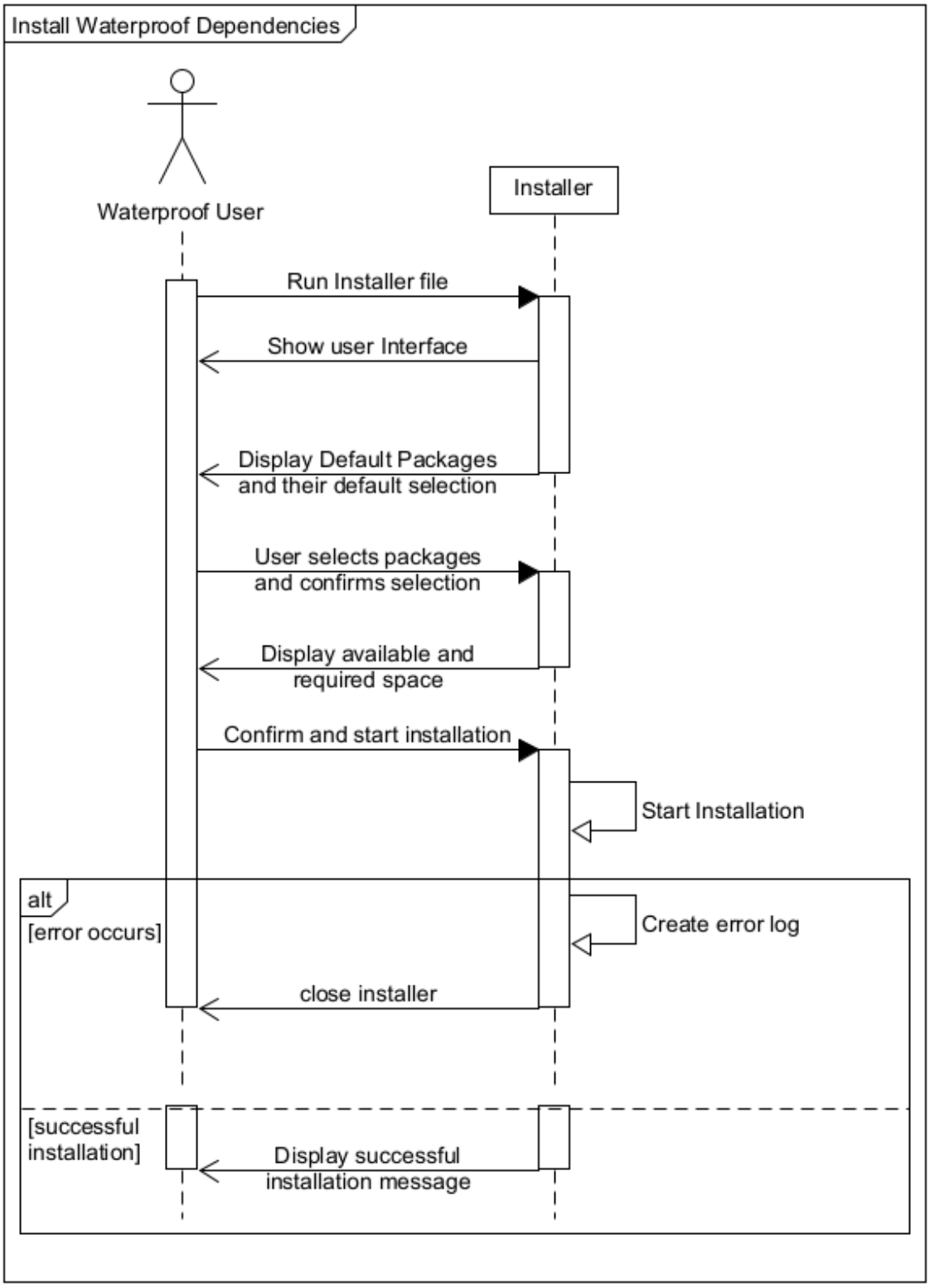


Figure 13: Installation of the Waterproof dependencies

If the user wants a custom package list installed, the user specifies which software packages should be installed by clicking a checkbox in front of the package name. Otherwise,

the configured default packages will be selected. Then, the user starts the installation procedure. The system notifies the user about how much space is needed for the installation. The system starts installing the selected dependencies. If no error occurs, the installer finishes and notifies the user of a successful installation. Otherwise, the installer closes, and an error is stored inside an error log file accessible by the user. This procedure concerns the A.7, A.8, and A.12 use cases of the URD [4].

3.3.3 Installing Waterproof and its dependencies

Figure 14 shows how the complete installation process of Waterproof and its dependencies takes place. The Dependencies installer is run first using the procedure described in the previous subsection (cf. 3.3.2), and then the Waterproof installer is run. If either the dependency installer or the Waterproof installer fails, then the entire installation procedure fails.

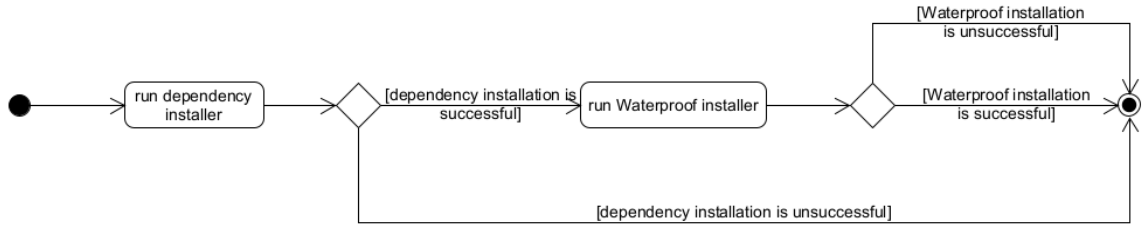


Figure 14: Complete installation of Waterproof and its back-end dependencies

3.3.4 Updater application

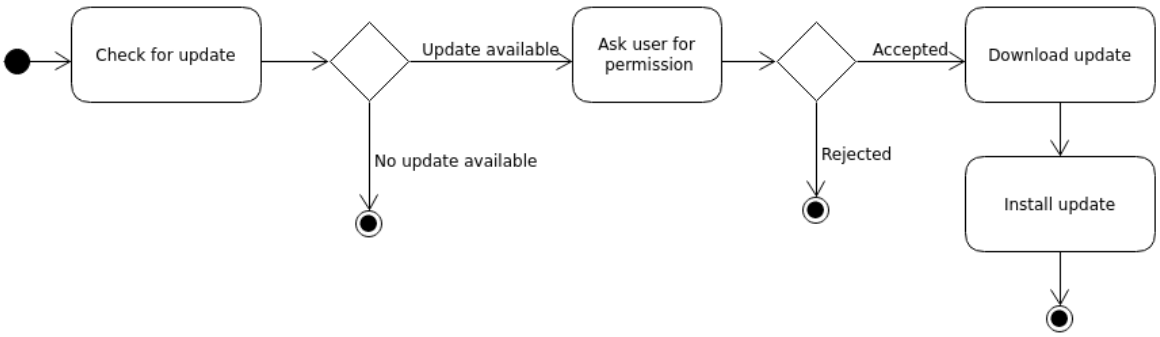


Figure 15: Updater steps

The Waterproof updater is based on the Electron `AutoUpdater` component, which is included in the Electron framework used by Waterproof. For more information, refer to the documentation corresponding to the Electron-updater package (cf. [10]).

When the Waterproof application is started, the updater checks whether a new update is available on the updater infrastructure. If this is the case, the user is provided with the possibility of updating immediately or postponing the update to a later moment. If the user accepts the update, the updater will start downloading the update from the updater infrastructure, and once complete, install them on the user's system. If the user postpones the update, the updater component is stopped, and the next time the Waterproof application

is started, the updater will once more provide the possibility of updating to the user. The situation is illustrated in Figure 15.

3.3.5 Uninstaller application

The uninstaller encapsulates a single state. When the user runs the uninstaller application Coq, all the Waterproof dependencies are uninstalled. However, this does not uninstall the Waterproof front-end, which is a separate application.

3.3.6 Abstract Syntax Tree parser

The state diagram for the AST-parser consists of only two states as can be seen in Figure 16.

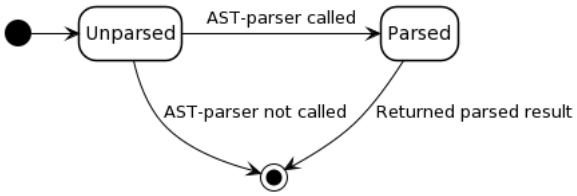


Figure 16: Parsing steps

Until the AST-parser is called, the input stays unparsed until the program is terminated. However, once the AST-parser is called, the state changes to parsed one, and the resulting CoqAST data structure is returned to the caller. It is important to note that the AST-parser is only called when the "Output All ASTs" button is pressed.

3.3.7 Dynamic state interaction of the tactics

Waterproof's tactic library only interacts with Coq (as it is technically a pure Coq library). Within Coq, there are only three types of states:

1. **Global definitions**: global bindings of an identifier to a definition. They include previously proven lemmas and theorems, definitions of (inductive) types, functions and axioms. They can only be created and modified using Vernacular commands. Since tactics can only be used as Gallina code, none of the tactics will affect global definitions.
2. **Global variables**: $L_{\text{tac}}2$ supports the use of global variables. Such variables are a mutable literal such as an integer or a list. The tactics library uses three global variables, all for configuring the behaviour of automated tactics. Users cannot directly change the values of the variables. Instead, they can be configured by importing pseudo-modules (see Section 3.5.4.1 for the interface of these pseudo-modules).
3. **Proof state**: Unlike the global state elements described above, the *proof state* exists only locally within the body of the proof of a single specific lemma. Recall that the proof state consists of two parts:
   (a) A set of hypotheses. These hypotheses are propositions that are assumed to hold, propositions that have been proven locally (within the scope of the body of the proof), and assumed variables. These hypotheses do not exist outside of the proof body. Each hypothesis is identified by a so-called *identifier*, which is analogous to a variable name.

(b) An ordered sequence of *goals*. Each goal is a proposition that must still be proven to finish the proof of a lemma. A proof begins with a single goal, but it is possible to split this goal into smaller sub-goals[2] or introduce sublemmas (that come with their own goal). Most operations are done on the goal that is currently on top (the 'topmost' goal).

The set of hypotheses may also contain definitions. Definitions are simply identifiers that can be used as a substitute for an expression, very much like a macro. They do *not* witness the existence of a term as defined in the definition. Note that the proof state changes as a proof proceeds (in a more practical sense, each line of proof code can have a different proof state. This new state is the result of executing the previous tactic on the previous proof state.). Furthermore, Gallina commands (most notably, tactics) can change the proof state. Hence, the tactics of the tactics library interact heavily with the proof state.

The different tactics of the library affect the proof state in various ways. These interactions are further described below, where tactics have been grouped according to their behaviour regarding the proof state

The tactics that employ automation use the same automation algorithm. For this reason, the automation will be described separately.

3.3.7.1 Variable Introduction The Waterproof library offers several tactics to introduce a new variable or a hypothesis into the proof state. All these tactics add a new hypothesis to the proof state but may also change the top-most goal while doing so.

- **Assuming any arbitrary variable**: the default method in logic to prove a proposition that contains a $\forall$-quantifier is to assume any arbitrary free variable of the same type as the bound variable. For example, to prove $\forall x \in \mathbb{N}, x > 0$ implies $x > x - 1$, one introduces an arbitrary natural number x such that $x > 0$, and shows that it follows that $x > x - 1$. The tactic `Take` can be used to add such an arbitrary variable to the set of hypothesis while stripping the $\forall$-quantifier from the goal.
- **Assuming a premise**: to prove an implication of the form $A \Rightarrow B$ (where A and B are some logical propositions), the default method is to assume A holds, and that B follows from this assumption. The `Assume` tactic can be used to add such a premise A to the set of hypotheses. When assuming a premise, the goal is changed to only the conclusion (i.e. from $A \Rightarrow B$ to the sole proposition B).
- **Picking a specified variable**: The tactic `Choose` can be used on a goal with a $\exists$-quantifier. Let the bound variable of the $\exists$-quantifier be m, then `Choose` adds m as a hypothesis to the proof state, and replaces the goal by the body of the $\exists$-quantifier (i.e. the $\exists$-symbol is removed, and only the inner statement remains). The tactic `Choose such that` implements logical $\exists$-elimination on a hypothesis. It can be used if there a hypothesis h of the form $h := \exists[x \in S \ : \ p(x)]$ (where $p(x)$ is some proposition on x). The tactic will add $x \in S$ and $p(x)$ as two new hypotheses to the proof state and leaves h unchanged.

  A concrete example where `Choose such that` can be used is the following: let s as the definition of convergence of a sequence of real numbers to a real number a. Call

[2] The terms 'goal' and 'sub-goal' will be used interchangeably. The Coq system makes no distinction between goals and sub-goals, but in some context, it is convenient to explain how a goal can be divided into smaller ones.

this sequence $(a_n)_{n\in\mathbb{N}}$. More formally,

$$s : [\forall \varepsilon > 0, \exists N \in \mathbb{N}, \forall n \geq N \ : \ |a_n - a| < \varepsilon] \, .$$

Fixing ε to a particular value, e.g. setting $\varepsilon = 1$, yields the hypothesis

$$h : [\exists N \in \mathbb{N}, \forall n \geq N \ : \ |a_n - a| < 1] \, .$$

The `Choose such that` tactic can be used to add both (1) the number N and (2) the hypothesis $h' := [\forall n \geq N \ : \ |a_n - a| < 1]$ to the proof state.

- **Defining a new variable**: the tactic `Define` allows binding an identifier to a definition. It only adds an identifier mapping to the definition to the proof state. Note that this definition is not a hypothesis that witnesses a term as defined in the definition.

3.3.7.2 Informative tactics The tactics `We know that`, `We need to show that` and `Help` do not affect the proof state. Instead, they read the proof state (and, if needed, compare it to user-proved arguments) and print a help message.

3.3.7.3 Goal splitting tactics There exist goals (propositions) that are generally 'split' into sub-goals (for instance, a conjunction of simpler propositions).

- Goals of the form $A \Leftrightarrow B$ (for some propositions A and B) are often proven by deriving $A \Rightarrow B$ and $B \Rightarrow A$. For this purpose, Waterproof offers the tactic `We show both directions`.
- Goals of the form $A \wedge B$ can be approached by proving A and B separately. The `We show both statements` tactic can be used to replace such a conjunctive goal by two sub-goals (which the two operands of the conjunction).
- Some goals can be proven with a case distinction. For example, $f(x) > x$ might be proven separately for the case that $x > 0$ and for the case that $x \leq 0$. The tactic `Either` can be used to make such a case distinction. Note that each of the new sub-goals add their own hypotheses (i.e. $x > 0$ and $x \leq 0$ respectively) to the proof state when they become the topmost goal.
- Goals concerning recursive definitions are often proven by induction. One first proves a base case, and then an inductive case that uses an induction hypothesis. The base case and the inductive case act as sub-goals. The induction goals can be created with the tactic `We prove by induction on`. Note that after a user proved the base case, Coq will remove it from the proof state and the inductive step automatically becomes the new goal. Coq will also add an *induction hypothesis* to the proof state as a hypothesis.

3.3.7.4 Applying previous results It may occur in a proof that one has derived a proposition A, while the goal or another local hypothesis states $A \Rightarrow B$ (for some proposition B). In this case, the tactic `Apply` can be used to replace the implication with the sole proposition B. This is equivalent to logical $\Rightarrow$-elimination.

Note that this can be used for both a goal or a hypothesis of the form $A \Rightarrow B$, as long as the hypothesis A is present in the proof state.

Alternatively, `Apply` can also be used to derive the hypothesis B if there exist both a previously proven lemma of the form $A \Rightarrow B$ and a hypothesis A.

3.3.7.5 Automation Many of the more advanced tactics make use of the subroutine `waterprove`. `waterprove` relies on Coq's built-in automation features. These automation features can make many automatic changes to the proof state in a single command to prove a sub-goal. They rely on the built-in Gallina tactic `auto` (or the related tactics `eauto` and `intuition`). These built-in functions essentially split the goal into several sub-goals, and try to resolve each sub-goal by using a hypothesis or a lemma. This lemma can originate from one of the hint databases or be proven earlier in the same proof-script. If any of the sub-goals cannot be proven, then the automation aborts and undoes its changes to the proof state. The `waterprove` function wraps this behaviour in a function that (1) uses different variants of `auto` in sequence and (2) raises an error if none of the variants could prove the current goal. Note that the automation will fail if it cannot *find* a proof for a goal; this may also occur if the goal is logically provable (with more steps and/or more hint databases).

3.3.7.6 Introducing New Claims When proving a theorem or a lemma, it might be convenient to prove a smaller claim during the proof. In Coq this is done by adding claim as an additional goal. After proving this goal, the claim will be added as a hypothesis to the proof state Waterproof offers the following tactics for this purpose:

- `We claim that`: this tactic adds a claim as a new topmost goal. The original topmost goal remains on the second position in the sequence of goals. It is left to the user to prove the claim; if they succeed, it will be added to the proof state as a hypothesis.
- `It holds that`: this tactic introduces a claim as the new topmost goal. It immediately applies the `waterprove` function to prove this goal automatically. An error will be raised in case `waterprove` fails. From the perspective of the user, either a new hypothesis (containing the claim) is added to the proof state, or an error is raised.
- `It suffices to show`: this tactic can be used to 'try out' if a claim would be able to solve the current goal. This tactic directly adds the claim as a new hypothesis and attempts to prove the old goal using the claim via `waterprove`. The old goal will be removed in case of success, but the claim will be added as a new goal. From the user's perspective, the tactic simply removes the old goal, and adds the claim as a new goal. In case of failure, an error will be raised.

  Note that from a logical point of view, this method does exactly the same as `It holds that`, only the goals are in a different order. More practically, `It holds that` proves the claim automatically, while `It suffices to show` proves the old goal automatically.

3.3.7.7 Rewriting Goals and Hypotheses Waterproof offers several tactics that can be used to rewrite a goal or a hypothesis. They can be grouped into four kinds of rewrite operations:

1. Direct substitution of equivalence (e.g. if one knows that $x = y$, then $x > 10$ can be rewritten to $y > 10$).
2. Applying transitivity on inequalities (e.g. if one knows that $y < x$, then $x < 10$ can be rewritten to $y < 10$).
3. Replacing a function by its definition (e.g. if one knows that $f(x) = x^2$, then $a \cdot f(x) + b = c$ can be rewritten to $a \cdot x^2 + b = c$).
4. Replacing a conjunction hypothesis with two hypotheses (i.e. if one knows that $A \wedge B$, then it follows that one knows A and B).

These different methods of rewriting have different implementations regarding state dynamics, which will be described further below.

Rewriting equalities The tactics `Rewrite using` , `Rewrite <- using` , `Write using` and `Write goal using` use an equality to rewrite a target. They all affect the state dynamics using the same general algorithm, given in Algorithm 1. Each tactic takes at least a literal equality proposition ('p') as an argument (i.e. an expression of the form $A = B$). Some of these variants also refer to a target hypothesis t to rewrite; the other variants rewrite the topmost goal (in this case, t will refer to the goal). The algorithm works as follows: first, it uses the automation function `waterprove` to verify that the input literal expression p holds (or throws an error if it does not), and adds it to the proof state as a hypothesis h. Then this hypothesis h is used to rewrite the target t (and replace it with the rewritten version in the proof state). Note that creating h is necessary for Coq: the system does not allow rewriting expression by using an unproven literal proposition. Finally, h is removed from the proof state. The user only sees the changes in t, as h is already removed before the algorithm terminates.

There is also the more general tactic `Write as`. This can be used to rewrite a hypothesis directly into another form (i.e. replaces the old hypothesis with the rewritten form while keeping the same identifier). This tactic relies on Coq's built-in ability to verify whether the rewrite is logically valid, and in practice, it can only be used for trivial equivalences (e.g. if $h := 1 + 1$ then it can be rewritten to $h := 2$ without the specific use of a proposition).

Algorithm 1:
REWRITEEQUALITY(p, t)

`Input:` p: a literal proposition of the form $A = B$.
t: a target, either a hypothesis or the topmost goal.
`Output: None`

```
1  begin
       /* This is done in order to verify the provability of p          */
2      Try to prove p using WATERPROVE(p) ;
3      if the call to WATERPROVE failed to prove p then
4          Raise an error. ;
5      else
6          Add p to the proof_state with identifier h ;
7          if t contains no occurrences of the term A then
8              Raise an error. ;
9          else
               /* Coq does not allow rewriting without using an available
                  hypothesis or lemma, hence h is strictly needed for the
                  following:                                             */
10             t' ← t with all occurrences of A replaced by B (via applying h);
11             Remove t from the proof_state;
12             Add t' to the proof_state;
13             Remove h from the proof_state;
```

Rewriting inequalities If one has a goal of the form $A > E$, and one knows that $A > B > C > D$, then it is logically sufficient to show that $D > E$. The tactic `Rewrite inequality` can be used for this purpose. This algorithm recursively changes the goal from $A > E$ to $B > E$

to $C > E$ to $D > E$. The tactic uses automation at each step to prove each required relation $A > B$, $B > C$, etc. After proving a single relation, it directly applies built-in transitivity lemmas. Note that from a user's perspective, the goal in the proof state is directly changed from $A > E$ to $D > E$. This tactic can also be used for the relations $=$, $<$, $\leq$ and $\geq$.

The tactic will fully prove and remove the current goal if the last step $D > E$ is also part of the input chain (i.e. the complete input chain is $A > B > C > D > E$). If the last step is not part of the input (i.e. only $A > B > C > D$ is supplied as an argument), then the goal is only rewritten to $D > E$.

A more general description of the procedure followed by `Rewrite inequality` is given in Algorithm 2.

Algorithm 2:
REWRITEINEQUALITY($chain$)

`Input:` $chain$: a chained proposition of the form $e_1\ R_1\ e_2\ R_2\ \ldots R_{n-1}\ e_n$ where each $e_i \in \mathbb{R}$ and each $R_i \in \{=, <, \leq, >, \geq\}$
`Output: None`

```
1  begin
2      for i = 1 to n − 1 do
3          p ← e_i R_i e_{i+1} ;
           /* p needs to be proven before it can be added to the proof state and
              referred to. Automation (waterprove) is used for this purpose.  */
4          Try to add p as a new hypothesis h to the proof state by proving by using
            WATERPROVE(p) ;
5          if the call to WATERPROVE failed to prove p then
6              Raise an error ;
7          Let the goal be g := g_1 R_g g_2 ;
8          if R_i and R_g are both inequalities and in different directions then
9              Raise an error ;
10         else if g_1 ≠ e_{i+1} then
11             Raise an error ;
12         else
13             if g ≡ h then
14                 Remove g from the proof_state;
15                 Remove h from the proof_state;
16                 return;
17             else
                   /* ≤ is weaker than <, which is weaker than =.          */
18                 Let R̂ be the logically least strong of R_i and R_g ;
19                 g ← e_{i+1} R̂ g_2 ;
20                 Remove h from the proof_state;
```

Expanding function definitions The tactic `Simplify what we need to show` attempts to replace as many functions in the topmost goal by its definition as possible, or even evaluate the return value where possible.

The tactics `Unfold` and `Expand the definition of` can be used to replace all occurrences of a specific function by its definition. This tactic can be applied to a goal or a hypothesis.

Destructing a conjunctive/disjunctive hypothesis. If there is a hypothesis h of the form $h := A \wedge B$ or of the form $h := A \vee B$, then the `Because` tactic can replace h:

- in the first case, ($h = A \wedge B$), with two hypotheses $h_1 := A$ and $h_2 := B$.
- in the second case ($h = A \vee B$), with a case distinction (the first one with the hypothesis $h_1 := A$ and the second one with the hypothesis $h_2 := B$), but also requiring a proof of the same goal in each case.

3.3.7.8 Proof Finishing Tactics Waterproof provides tactics that can directly finish the topmost goal, given that the topmost goal is sufficiently easy. If there was only a single goal, these tactics would finish an entire proof, but in other cases, they will only prove the topmost sub-goal of the goal list.

Proving one sub-goal out of multiple goals will remove it from the proof state, together with any hypotheses bound to it. Other goals and more general hypotheses will remain in existence. Furthermore, new hypotheses may be added to the proof state when a new goal becomes the topmost goal (for example, for an inductive proof, the induction step comes with an *induction hypothesis*. This induction hypothesis is only added to the proof state as soon as the inductive-step sub-goal becomes the topmost goal in the goal list).

The following specific tactics can be used to finish proofs:

- `This follows by reflexitivity`: removes the topmost goal if it is of the form $A = A$.
- `This follows by assumption`: removes the topmost goal if there is a hypothesis with the exact same statement as the goal. A variant of this tactic is `Then ... follows by assumption`. This variant does the same, but also checks if the user-supplied the correct goal.
- `This concludes the proof`: attempts to prove (and remove) the topmost goal via the automation procedure `waterprove`.
- `We conclude that`: same as `This concludes the proof`, but requires the user to state the current value of topmost goal explicitly. It raises an error if a wrong expression is given.

All these tactics raise an error in case they fail to prove the topmost goal.

3.3.7.9 Sets tactics. A particular set of tactics are the tactics for proving identities with sets. Two types of identities are covered:

- Automatically proving a trivial set of equations, by using the `waterprove` automation function and other Gallina automation functions, such as `firstorder`. The sets identities in question are those of the form $A \cap B = B \cap A, A \setminus \emptyset = A$, etc. This is done via the tactic `This set equality is trivial.`
- Splitting the proof of a general set equality of the fort $A = B$ into its two respective parts given by the extensionality axiom (i.e. proving that $A \subseteq B$ and $B \subseteq A$). This is done by the tactic `We prove equality by proving two inclusions.`

3.3.7.10 Contradiction tactics. All the tactics we have presented so far were about *direct reasoning*. This means that such a particular tactic is used to attempt a direct proof of a specific goal or sub-goal.

However, we have also implemented two particular tactics that work in a completely different manner. These are the tactics that use the concept of *reductio ad absurdum*. The tactic `We argue by contradiction` is used so as to start a proof by contradiction by introducing the propositional negation of the current goal as a hypothesis to the proof state.

The tactic `Contradiction` is used to conclude a proof by contradiction (once a contradiction has indeed been derived), removing the current (top-most) goal.

3.4 Data model

In this section, the data model for each part of the project is discussed.

3.4.1 Persistent Data in the Installer and updater

The installer and updater applications do not use any persistent data.

3.4.2 Persistent Data in the Abstract Syntax Tree parser

There is no persistent data concerning the AST parser. Everything is done dynamically. Therefore data-model design is not relevant to the AST section.

3.4.3 Persistent Data in the Tactics Library

The tactics library does not use persistent data. Only the hints in the hint databases could be interpreted as persistent information.

Recall that hints databases are no tabular databases but simple collections of hints. Hints are theorems or lemmas, and their full proof must be given before they can be part of a database. Such proofs could be interpreted as persistent data, although it is arguably more convenient to regard them as conventional 'code'. Note that in Coq, proofs must always be given to refer to a lemma: the code does not compile otherwise. This guarantees that all lemmas added to a hint database are valid.

3.5 External interface definitions

In this section, the external systems that interact with the Waterfowl project as a whole are described.

3.5.1 Installer

The installer uses a configuration file that specifies which opam packages need to be installed. It also specifies Github repositories containing Coq libraries to be installed. In addition, each package has an option to be installed by default in the installer or not.

The configuration file contains two types of lines.

Firstly, for custom Github package six items need to be specified; each of them between quotes (e.g. ""):

1. Package name: the name of the package
2. Installed by default in installer (SELECTED/UNSELECTED)
3. Relative package location: the path location in windows format relative to the Opam switch

4. Package description: textual description shown in installer
5. Github owner: repository owner
6. Github repository: name of the repository

The opam switch folder is the folder containing all files installed by opam and is located in the folder `/.opam`.

The syntax is then:

```
GITHUB "[package name]" "[selected]" "[relative package location]" "
   [package description]" "[Github owner]" "[Github repo]"
```

The second type of line is for Opam packages. The following argument should be specified for this:

1. Package name: the name of the Opam package
2. Installed by default in installer (SELECTED/UNSELECTED)

The syntax is:

```
    OPAM "[package name]" "[selected]"
```

3.5.2 Updater

As mentioned before, the updater is completely based on the electron-updater library. The Waterfowl project has not added any interfaces, and as such, the external interface definitions are entirely determined by this library. See the documentation corresponding to the electron-updater package for more information.

3.5.3 Abstract Syntax Tree parser

The AST parser does not depend on any external interface. Furthermore, no interface for direct communication with the AST-parser is defined. The only possible interaction is using the parsing process results as input for an external functionality.

This is the current behaviour for the flattening operation. To flatten an AST, an instance of the FlattenVisitor class has to be created, to which the AST is referenced.

3.5.4 Tactics library

The tactics library provides interfaces to other systems on various levels:

1. An interface for importing library files in a Coq script (as needed by the end-user).
2. A file structure. This is mainly of interest to the developers and maintainers.
3. Compilation instructions for compiling the entire library.
4. Instructions for opam (cf. [15]) to download and install the library.

3.5.4.1 Importing the library in Coq If successfully installed, the tactics library can be imported as any other Coq library. In particular, to import all tactics, one can run the following command:

```
    Require Import Waterproof.AllTactics.
```

To enable alternative notations for built-in syntactic elements of Coq[3], the following commands can be used:

```
Require Import Waterproof.notations.notations.
Require Import Waterproof.definitions.set_definitions.
```

It is technically also possible to import a specific tactic by using the relative path within the library to the file that implements the tactic. However, this does not seem to offer any practical advantage to any user. The `AllTactics` import works as a macro that performs the same behaviour as importing all the individual tactics separately.

Configuring automation The behaviour of the automation function `waterproof` (on which many tactics depend, see Section 3.3.7) can also be configured by using the

`Require Import`

command. Waterproof offers pseudo-modules that only change the global state, but do not load any other definitions. For example, a user can set the 'search depth' setting to the value 5 by running:

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

The following sets of pseudo-modules are available:

- Search depth: all available imports with prefix `Waterproof.set_search_depth.` Currently the available import are `To_1`, `To_2`, ..., `To_5`.
- Hints databases used: users can add more hint databases to the set of databases used by `waterprove` by importing any of the pseudo-modules with a prefix

  `Waterproof.load_database.`.

  Currently the available imports are `AbsoluteValue`, `Additional`, `Exponential`, `Other`, `Multiplication`, `PlusMinus`, `reals`[4], `RealsAndIntegers`, `Sets`, `SquareRoot`, `ZeroOne`.

  There is also a special module `All`, which imports all the other hint databases. Similarly to the `AllTactics` import, it only acts as a shortcut for importing each individual database separately.
- Wildcard-flag: advanced users can use the commands[5]

  ```
  Require Import Waterproof.load_database.EnableWildcard.
  ```

  and

  ```
  Require Import Waterproof.load_database.DisableWildcard.
  ```

  to respectively enable and disable the use of databases by `waterprove`. Disabling the use of databases causes `waterprove` to use all databases that Coq can find in all imported modules. This feature is disabled by default.

[3] This feature was already part of the $L_{\text{tac}}1$ implementation of the tactics library. It was not rewritten for the new library, as it only uses Vernacular commands instead of $L_{\text{tac}}1$ commands.

[4] The `reals` database does not begin with a capital letter, to avoid confusion with Coq's build-in library `Reals`.

[5] The name 'Wildcard' stems from the Coq notation for using all databases: the specific statement is `auto with *`. The '`*`' symbol is often referred to as a 'wildcard' in software systems where it is a shorthand for 'all possible arguments'.

- Intuition-flag: similarly to the wildcard-flag, advanced users can turn the use of the tactic called `intuition` (a variant of Coq automation that `waterprove` optionally uses) on and off. The respective commands are

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

and

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

This feature is disabled by default.

3.5.4.2 File structure. The tactics library is implemented as an organised collection of text files with the `.v` extension. This is the standard file format used by Coq (and hence for Gallina, Vernacular and Ltac2 code).

The organisation of the repository is as follows:

- The root directory contains a sub-directory named `waterproof`, the file `_CoqProject` (described below), a `Makefile` (also described below), a licence file and markdown file `README.md`. The `README.md` contains a textual summary of the purpose and use of the library.
- The directory `waterproof` contains several sub-directories and `.v` files (containing auxiliary functions). The most important sub-directories are
  - `tactics`: contains the implementations of all tactics.
  - `test`: contains the unit-tests of all the tactics and of the automation configuration.
  - `waterprove`: contains the implementation of the `waterprove` automation function .

3.5.4.3 Compilation instructions The root directory of the library's source code (see [17]) contains two files that prove instructions for compiling the source code: a file named `_CoqProject` and a file named `Makefile`. The `_CoqProject` file describes two things: he logical name that should be given to the directory `waterproof` within Coq's import system, and an ordered list of the files that need to be compiled. The `Makefile` ensures that the program *GNU make* [11] is able to read the `_CoqProject` and compile the complete library. From a user's perspective, all that is needed to install the library is to run the command `make` from the root directory of the library.

3.5.4.4 Instructions for opam The tactics library can also be installed via opam. The opam software can be configured to recognise packages from the opam-coq-archive ([16]). For this purpose, there exists a file called

`/released/packages/coq-waterproof/coq-waterproof.1.0.0/opam`

in the opam-coq-archive. This file adheres to a format that opam can interpret. It describes from which URL the source code can be downloaded and how the source code can be compiled.

3.6 Design rationale

In this section, various design decisions made during the development of the Waterfowl project will be described and argued in contrast to the alternative technologies and/or solutions.

3.6.1 Installer Framework

Multiple options have been considered when choosing an installer framework for the waterproof dependency installer. Those options include Installshield, QT, Innosetup, WiX, NSIS, as well as the possibility of creating a custom installer without a framework. Due to an excessive price, the popular option Installshield will not be considered. A custom application can have all features available at the cost of having a complex development process. We think a custom application would have been too much for our needs.

An in-depth analysis of the other installer frameworks was made. This comparison will be showcased through the following table:

	QT	Innosetup	WiX	NSIS
32 bit support	✔	✘	✔	✔
Learning curve	Normal	Easy	Hard	Easy
Cross platform	✔	✘	✘	✘
Uninstall built in	✔	✔	✔	✔
License	(L)GPL dual-licensed	BSD style license (free to do)	MS-RL (LGPL like)	Zlib license (free to do)
ARM support	Limited	✔	✘	untested/not native
Updating component	✔	✘	✘	✘
Extra info			Creates .msi	Used by OCaml, Coq

Figure 17: frameworks for Waterproof dependencies installer

QT is a cross-platform framework while the other three frameworks, namely Innosetup, WiX, and NSIS [13] work only on Microsoft Windows. Although QT is cross-platform, the installation code would not be portable since the installation of all modules is specific to the Operating System it is used on. But there is also a large part of the installer that could be reused if a Linux or MacOS version was necessary for the scope of creating an installer for multiple platforms.

Furthermore, due to their similarity, licensing was not a deciding factor in our choice. Innosetup is a more modern framework that does not support 32 bits. It is the only framework with full support for the ARM architecture. Uninstall is a feature built into each of the frameworks. The learning curve for each framework is quite different. With WiX being the hardest

to grasp and Innosetup and NSIS being relatively easy. The complexity of WiX is partly because it is feature-rich and is the only framework creating `.msi` files. For reference, a `.msi` file generally is smaller and more robust than a `.exe` file. It is a database that is then used by a Windows service to install the software. It isn't easy to install multiple `.msi` files from a `.msi`, and the flexibility of the installation process is reduced at the cost of a more robust implementation. Although possible to create a `.msi`, another installer is preferred to avoid the aforementioned problems. The QT install framework has an updating and maintenance component, which would significantly speed up the development of an updater.

The most important factor when comparing the four frameworks was that the coq-platform (cf. [8]) already uses the NSIS framework for the creation of its Coq installer. With all the other frameworks, every new release of the coq-platform could potentially require changes to the installer created, while the option of using NSIS in conjunction with the already existing method of generating an installer would take less time and would be more consistent across multiple releases of the coq-platform. Furthermore, the client was also interested in using NSIS for this reason. Since the benefits of using other installer frameworks did not outweigh the benefit of using NSIS coupled with the already existing code of the coq-platform, the installer was created using the NSIS framework.

3.6.2 Updater Framework

For the updater framework, the Electron-updater package was chosen. The Qt installer framework and Winsparkle could also be used as the updater framework. For the Qt installer, this is difficult without using the framework to build the installer as well. Winsparkle does not offer good integration with Electron since it has limited customizability. The electron-updater integrates directly in Waterproof and provides a seamless experience. This makes it the best option. One of the downsides is the limited documentation, but the excellent integration with Waterproof outweighs this downside.

3.6.3 Abstract Syntax Tree parser

For the AST parser, we opted to develop a strict implementation based on pattern matching and concrete classes as opposed to more automated mechanisms such as

`js_of_Ocaml`.

Given the data provided by SerAPI, we convert the s-expressions into a nested array structure which is then traversed and matched with the appropriate CoqType.

The parser was developed using the TypeScript programming language.

In the following subsections, we will present the rationale for these choices.

3.6.3.1 Choice of programming language As the Waterproof application is an ElectronJS project it was mandatory to use a language that can closely interact with JavaScript. This leads us to three possibilities, either using JavaScript, a language that interfaces with it via WebAssembly, or compiles to it.

While JavaScript is a very popular programming language, it has many features and particularities that can be quite unintuitive and introduce issues. Furthermore, it is a dynamic types language, so errors can only be discovered at run time.

WebAssembly is a promising technology that allows languages such as C++ and Rust to be compiled to a format usable by JavaScript. However, the technology is still experimental,

and significant changes would have to be introduced to the build process of Waterproof to support it.

Thus, we opted for the latter of the three, choosing TypeScript as our implementation language.

TypeScript acts as a syntactic-superset of JavaScript, augmenting the language with a robust type system. This feature allows for look-ahead static validation of code, preventing the appearance of hard to notice bugs and discouraging harmful coding patterns resulting from JavaScript's loose language design.

A significant advantage of using TypeScript is that it can be easily integrated with existing code, as any JavaScript snippet is also valid in TypeScript. Moreover, it has excellent integration with existing tooling used by web developers daily, so it is easy to integrate within the current codebase.

Hence, we decided to adopt TypeScript for our programming language.

3.6.3.2 Preference for static class structure

There are many ways to create a parser for a given grammar. Still, the main difference we are interested in is the level of contextual information the parser has about the underlying input it is converting.

Thus, we distinguish between two types of parser techniques:

- dynamic parsing, in which the conversion from an s-expression to a use-able data structure is done automatically
- static parsing, which relies on domain understanding of the types produced by Coq and SerApi

While dynamic parsing is a very attractive prospect, and there are some tools designed to support the conversion from OCaml to JavaScript, namely the `js_of_ocaml` and ReasonML projects, the quality of the resulting output is lacking. Furthermore, upon conversion, the data structures from OCaml produce code which can sometimes be hard to understand and would still need to be decorated with custom functionality. Moreover, part of the Coq and SerAPI code would have to be embedded directly into Waterproof for these conversion strategies to work.

Thus, we opted for a static parser approach, which relies on understanding the data structures as Coq specifies them. This process is more involved initially, as familiarity with functional programming alongside experimentation is required. However, this allows for greater flexibility in terms of development, such as enforcing the same coding standards as the rest of the Waterproof project.

3.6.3.3 Using the visitor pattern

The visitor pattern is an OOP behavioural pattern that embodies a similar design principle as its popular functional-programming counterpart, pattern matching.

It allows for dynamic operations over complex, and inherently recursive data-structures without changing the objects on which algorithms operate.

Thus, it is straightforward to implement new behaviour isolated from the underlying data structure, such as implementing a pretty-printer or an export to HTML functionality.

Alternatively, for each new feature someone wants to add that leverages the AST-parser, all the classes inheriting CoqType would have to be modified to accommodate, which breaks

many design guidelines and becomes error-prone and unmaintainable very quickly.

3.6.4 Tactics library

This subsection will discuss the design choices that govern the implementation of the new tactics library.

3.6.4.1 Choice of programming language The new tactics library was developed using $L_{\text{tac}}2$ in the proof assistant Coq. Note that there exists another software designed for mechanising mathematical proofs, such as `AGDA` (cf. [1]), which is an alternative for Coq. Coq and $L_{\text{tac}}2$ have been chosen for the following reasons:

1. The customer explicitly required a Coq library written in $L_{\text{tac}}2$.
2. The original library was written in Coq using $L_{\text{tac}}1$. It is more convenient to check if the new library offers the same capabilities when using the Coq system. For example, the customer suggested using proof-scripts that employed the previous library as acceptance tests for the new library (after minor modifications). These proof-scripts are written in Coq's Gallina language.

3.6.4.2 Modularity An essential objective of the new tactics library was to create a modular and extensible package. This was one of the main requirements of the customer but also more practical in implementation. For example, the automated tactics use the hints databases, which may be reconfigured in the future. Another example is that the test cases for each tactic rely on a reusable collection of assert statements.

3.6.4.3 User Feedback One of the customer's requirements was to improve the feedback that the tactics library offered to the user (with respect to the old library). This feedback consists of error messages and hints. As mentioned in Section 3.1.5, the tactics are designed to print many messages/throw many concrete errors, allowing the user to understand their mistakes and how the functions can be used.

3.6.4.4 Encourage Explicit Code and proofs Since Waterproof is mainly employed as educational software, the customer requested that the library encourages users to write readable proofs. For this reason, many of Waterproof's tactics are more verbose than build-in tactics. Many also require the user to be explicit about their actions and desired results (by default, Coq tries to infer the users intent from concise statements. This makes it very difficult for a user to read a proof-script without running Coq line-by-line).

3.6.4.5 Configuration via import system As described in Section 3.5.4.1, one can configure the behaviour of the automation routine (`waterprove`) at runtime by using the Coq built-in command `Require Import`. This command will be already familiar to the user, and they are not aware that the configuration modules are only pseudo-modules that change a global variable.

Exposing the users to $L_{\text{tac}}2$ would be the only other option to configure the variables at runtime. However, since it is generally assumed that the users are not experienced Coq users, requiring the use of $L_{\text{tac}}2$ seemed much too demanding and confusing. Note that one does not need to use or see any $L_{\text{tac}}2$ code to use any other tactic from the library.

3.6.4.6 Motivation for automation and hints databases Coq uses a strictly formal mathematical system, in which, for example, $x = y$ and $y = x$ are not the same terms. Of course, they are logically equivalent, and Coq includes a lemma that asserts this equivalence. However, it is very time-consuming to find each required lemma for equivalences that are generally considered trivial from a user perspective.

Waterproof speeds up the process of writing a proof by using the automation features that Coq offers for such 'trivial' steps in a proof. These automation features automatically try to advance a proof by using a backtracking algorithm that tries to apply lemmas from a given set of lemmas. They may fail to complete a complicated proof, but they are generally very suitable for finishing 'trivial' steps. These sets of lemmas used by automation are specified in the *hint databases*.

The set of hint databases needed may vary by the context of the proof. For example, lemmas for sets are only needed for proofs involving set theory. Not all databases are loaded by default, since using more databases than needed significantly increases the runtime of automated tactics (as explained in Section 4.4.1).

4 Feasibility and resource estimates

This final chapter provides an overview of the feasibility and resource estimates for each of the improvements made to the Waterproof application during the Waterfowl project.

4.1 Installer

The installer needs a Windows environment. But it is built by Github Actions on Github servers. Hence, only a basic computer with an internet connection is required.

4.2 Updater

The updater uses the Electron-updater [2] library, which is an official electron library. Although documentation is limited, it requires little code to have a working updater. Furthermore, limited code has to be maintained as most maintenance will be done in the library. The exact computer requirements are needed as discussed in Section 4.3

4.3 Abstract Syntax Tree parser

The AST parser integrates with the existing Waterproof project. Therefore, the requirements for the AST parser are the same as those for the original Waterproof project. While parsing does add a bit of overhead, it is noticeable, as confirmed by the unit tests. Furthermore, during development, we run the Waterproof software inside a virtual machine with only 6GB of RAM and a single CPU core. No performance change was noticed after enabling the AST-parser and the syntax highlighter.

4.4 Tactics library

The tactics library does not have strict resource requirements, it should work on any system that can run Coq. Compiling the library may take up to three minutes on a typical laptop [6].

4.4.1 Automation resource demands

The automation features are the only part of the tactics library that may use many resources. Executing an automated tactic typically requires a single CPU core to run at 100%, but the amount of memory and time needed is heavily dependent on the configuration of `waterprove`, as well as the current goal that the function must prove.

In particular, the runtime and memory use grow *exponentially* in the search depth and in the amount of hint databases used (the number of hints in a hints database also has an impact on this). However, recall that a user can configure these settings at runtime (see Section 3.5.4.1). This can be used to reliably constrain the resource demand, but only at the cost of making the automation less capable of solving its goals. When a small search depth and/or a small number of hint databases is used, the automation tactics may fail to prove larger steps. In this case, it will be up to the user to do more steps manually (e.g. by proving additional intermediate statements).

[6] tested on a system with an Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz processor and 8 GB of RAM), and does not require more than 2 GB of RAM.

A Packages installed by the Coq Platform

Package	Default	Module
coq-serapi	✗	SerAPI
coq-zfc	✗	ZFC
coq-unicoq	✓	general
coq-ext-lib	✓	general
coq-equations	✓	general
coq-equations	✓	general
coq-bignums	✓	general
coq-aac-tactics	✓	general
coq-mtac2	✓	general
coq-simple-io	✓	general
coq-quickchick	✓	general
coq-hott	✓	Homotopy Type Theory
coq-flocq	✓	Analysis and numerics
coq-coquelicot	✓	Analysis and numerics
coq-gappa	✓	Analysis and numerics
coq-interval	✓	Analysis and numerics
coq-elpi	✓	Elpi
coq-hierarchy-builder	✓	hierarchy builder
coq-mathcomp-ssreflect	✓	standard mathcomp
coq-mathcomp-fingroup	✓	standard mathcomp
coq-mathcomp-algebra	✓	standard mathcomp
coq-mathcomp-solvable	✓	standard mathcomp
coq-mathcomp-field	✓	standard mathcomp
coq-mathcomp-character	✓	standard mathcomp
coq-mathcomp-bigenough	✓	extra mathcomp
coq-mathcomp-finmap	✓	extra mathcomp
coq-mathcomp-real-closed	✓	extra mathcomp

Table 1: Default Coq packages installed using opam