Documentation and Formal Mathematics

Web Technology meets Theorem Proving

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen,
op gezag van de rector magnificus prof. mr. S.C.J.J. Kortmann,
volgens besluit van het college van decanen
in het openbaar te verdedigen op
dinsdag 17 december 2013 om 12.30 uur precies

door
Carst Tankink
geboren op 21 januari 1986
te Warnsveld

ISBN:978-94-6191-963-2

This research has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
The author was supported by the Netherlands Organisation for Scientific Research (NWO) under the project "MathWiki: a Web-based Collaborative Authoring Environment for Formal Proofs".

Contents

Preface

There is only one name on the front of this thesis, but as any researcher can tell you, you cannot do research and produce a thesis on your own. I have been especially fortunate to have a group of people who supported me, both professionally and personally, through the last four years.

First, I want to thank Herman and James, who have supervised me during my Ph.D. research, allowing me to plot my own course, but expertly guiding me with the right questions and requests for clarification.

This thesis was reviewed by a manuscript committee of three people, who sacrificed valuable time to read the manuscript and provide comments. Thank you, Henk, Michael and Makarius. When I defend this thesis, it will be before a committee of people in addition to the three above, who have likewise sacrificed their time: thank you, Johan, Theo, Freek and Julien. While I am thanking people who have read my texts over the years, I would like to thank the anonymous referees who have read the papers that formed the basis of this thesis. Their comments have improved the overall quality of the thesis.

During the ceremony, I will be supported by two wonderful "paranimpfen": thank you, Maya and Pim. I have done my research in collaboration with many others. I want to thank Josef especially for being a collaborator on the MathWiki project and writing papers with me, and Freek for planting the seed of what would become the Proviola. In addition, I am thankful to have worked together with Christoph and Cezary on two papers, who provided an external view on the research.

Back in Eindhoven, there were several people who have set me on an academic path: Jan Friso decided to hire me as a student assistant in his MCRL2 team, after a suggestion by Michel. Francien supervised me during my master's thesis research, and was the one who suggested I talk to Herman, causing me to start on this path.

In Nijmegen, I had the pleasure of sharing a room with many interesting and friendly people. One in particular stands out. Maya, thank you for being an awesome office mate, someone to talk to, complain to and drink tea (or cofee) with. Dick, Eelis, Evgeni, Hans, Jan, Saskia and Yowon, thank you as well for sharing the office with me over the years. Alyona, thank you for sharing train rides with me for a year, and reflecting on Dutch culture versus Russian culture.

The Intelligent Systems group gave me some very nice colleagues over the years. At the risk of forgetting someone and being reminded of on a later date, I want to thank Alexandra, Ali, Andrew, Bram, Dan, Daniel, Dimitrios, Elena, Fabio, Georgiana, Giulio, Janos, Jonce, Joris, Kasper, Lionel, Max, Nicole, Robbert, Saiden, Suzan, Tjeerd, Tom, Tom, Twan, Wout and Wouter

As a Ph.D. student, I was a member of the IPA research school, which provided several occasions for getting social with other Ph.D. students. I want to thank Tim and Meivan for organizing these events, and all members with whom I have talked, discussed and drunk.

Jeroen in particular has been a friend since we started university together and still is.

I have many happy memories of attending the Oregon Programming Language Summer School in 2011, and I met more people than I can thank by name. However, I want to thank Andrew for lending me his shirt when my luggage took a trip of its own with United, Sneha for being the happy presence of the school, and Emma and Joey for taking me bouldering for the first time.

Speaking of bouldering, as I mention in my “Stellingen”, it is the ideal sport for scientists. Thanks to the people at Monk Eindhoven for running the gym, and Daniel, Janina and Tim, Tom and Twan for making the sport social on occasion.

The second hobby that supported me socially during the Ph.D. is playing RPGs. Thank you Bart, Kris, Maikel, Mark, Peter and Rom for playing the campaigns with me.

Having the unconditional love of a dog to fall back on is something that is impossible to overestimate, and since dogs cannot read, I want to thank the people who allowed us to take them into our lives: Jannes and Anja, thank you for Ayla, even though she could not stay with us for long. Arthur and Paulien, thank you for Kaylee, who is lying next to me on the sofa as I write this.

Dad, mom and Bram, thank you for all the support you have given me over the years. Even if it was difficult for you to understand exactly what I was doing in my research, you occasionally tried :-)

Finally, there is one more person who needs to be thanked. Cora, my love, thank you for always being there for me, even when things were not going well for yourself.

Carst Tankink, Eindhoven, October 15, 2013.

Chapter 1

Introduction

Computer verified mathematics is a field on the border between computer science and mathematics: its aim is to develop tools and languages that allow mathematicians to write proofs that can be verified by a computer program (a proof checker or theorem prover).

The appeal of these programs is that they improve trust in a proof: they are typically designed around a relatively small kernel of logical rules, with which all proofs must be built: trusting a proof is reduced to trusting the (implementation of the) kernel, and whether the statement is translated correctly to the formalism of the proof checker. There are several proof checkers that have a different way of establishing trust, but the ones we consider in this thesis are mainly based on this architecture. For a more in-depth discussion on establishing trust in formal mathematics, we refer to Pollack [73] and Wiedijk [103]. A description of the social processes involved in accepting (or even creating) a computer-verified proof is given by Asperti, Geuvers and Natarajan [6]. Wiedijk also gives an overview [102] of the seventeen theorem provers that are most commonly used.

The improved trust is of benefit both for computer science, where it provides certification that software behaves according to a certain specification, and in mathematics, where proofs can become so complex and involved (possibly requiring computation) that it becomes difficult for humans to judge if a theory is 'correct'. An example of the former is an operating system kernel proven correct in Isabelle [54], where the verification guarantees that the kernel operates according to some specification and only that specification. For the latter, examples are the formal proof of the Feit-Thompson theorem [37], which is known for its length, and the proof of the Kepler conjecture [40], which uses computer programs to calculate a number of lower bounds; this proof, which is used as a case study in Chapter 6 was accepted with 99% certainty by the referees of the Annals of Mathematics and prompted a project to computer-verify the proof including the programs doing the calculations.

Apart from establishing trust, formal proofs can also be used to understand a theory: because a formal proof requires a more detailed explanation and explicitly named assumptions, writing a formal proof causes the author to consider the "design decisions" of the proof. While *writing* a formal proof can provide these insights, *reading* it is not necessarily useful: because of the greater amount of detail, and the different languages used (tied to a particular proof assistant), it can be difficult to see the main points of a proof. This inability for formal proofs to communicate their contents is not just a problem when trying to explain a proof to interested outsiders (such as traditional mathematicians or students), but also when communicating about a proof with collaborators. Moreover, there are no good tools that support the documentation of a formal proof: not only are the formal proofs themselves unreadable, it is also hard to add documentation that might provide insight to the reader. In this thesis, we present several tools and techniques that help in documenting formal proofs. Some of these tools are fully automatic, using the theorem provers to provide extra information about the proofs. Other tools are usable by authors of formal proofs, to document their efforts more easily.

The central theses of this work are:

- The language of formal proof is not suited to support human understanding, and efforts need to be made to improve this understanding.
- Instead of focusing the efforts on improving the formal languages or the proof checkers that require them, it is possible to support communication of formal proof by leveraging existing tools and technologies. In particular, by extracting information from the proof checkers and disclosing this information through web technologies.

There are several ways to improve the communication quality of formal proofs.

- Make the language of formal proofs similar to the mathematical vernacular (such as is done for the Mizar system and the Isabelle/Isar language).
- Make proof checkers more powerful, to support larger reasoning steps and more implicit bookkeeping. Support for this style of proving is given by, for example, the Sledgehammer tool for Isabelle. Typically, the stronger reasoning tools are also required to make the language more similar to the mathematical vernacular, as the tool is capable to deduce the implicit steps that are common in an informally described proof.
- Supply tools that allow users to describe their formal proofs as if they were informal, and giving automatic support where possible for rendering and displaying the formal proof for an audience that is not formally experienced: this is the approach taken in this thesis.

While there are many ways to provide tools for communicating formal proofs, the approach of this thesis is to support communication by disclosing proof checker information through 'recent' developments in web technology. By using the information that is already available in a formal proof and automatically transforming this in a generic format, we avoid two problems:

- We avoid having to adapt the internals of proof checkers in order to demonstrate new workflows. Instead, we build the workflows on top of exististing tools, and suggest how the tools can be improved to better support these workflows. This also means that there is less adaption involved in bringing different proof checkers to support the workflows.
- It does not require authors of formal proofs to write their proofs in a specific way. Because we impose no changes on the formal and informal input languages, existing documents can be supported by the tools developed for this thesis.

Specifically, we have looked at low-level documentation efforts through wikis, and adapted formal workflows to the wiki model. The discussed workflows are all supported through an onlne wiki platform for formal mathematics: the Agora system.

1.1 Wikis and other Web Technology

The last decade saw the rise of several interesting technologies hosted on the internet and based on improved browser technology.

Wikis The first technology is the *wiki*: an online collaboratively maintained database, that gives visitors an easy way to edit the pages they visit. Ward Cunningham developed the first wiki, aimed at software developers, in 1994, but wikis only hit the mainstream with Wikipedia, started in 2001, so we still consider it a recent development. The essence of a wiki, according to Cunningham and Leuf [63] is:

- A wiki invites all users to edit any page or to create new pages within the wiki Web site, using only a plain-vanilla Web browser without any extra add-ons.
- Wiki promotes meaningful topic associations between different pages by making page link creation almost intuitively easy and showing whether an intended target page exists or not.
- A wiki is not a carefully crafted site for casual visitors. Instead, it seeks to involve the visitor in an ongoing process of creation and collaboration that constantly changes the Web site landscape.

These principles are not entirely compatible with the principles of formal mathematics. In particular, inviting all users to edit any pages, including formal proofs, is a challenge: it is difficult to learn how to write such proofs, and a conceptual mistake in a proof can cause the entire database to become 'invalid'. Moreover, there are no stable tools that allow a user to comfortably edit a formal proof using just a plain Web browser: Kaliszyk's ProofWeb [50] was an attempt to create a web editor for formal proofs, but the implementation no longer works in modern web browsers, making it less comfortable to use, and the architecture does not lend itself to embedding in other systems (such as a wiki) very well.

On the positive side, because formal proofs have strong semantics, the topic association mentioned in the second bullet can be done semi-automatically: it is possible to export the definitions and lemmas of a formal proof, for reference in other texts. Moreover, in a wiki it becomes possible to refer to a lemma or definition that is not yet defined: the system will clearly mark that the target definition does not exist, and a user can add it. Obviously this is not enough for a formal system, which needs a well defined statement, if not a proof, to be able to use such an undefined lemma, but it does allow communication about formalizations that are in progress. Finally, the wiki is intended to incite collaboration, which we hope to extend from developer-developer collaboration to developer-audience collaboration.

Improved web technology Also in the last decade, mainly because of increased bandwidth in large parts of the world, the internet has seen the advent of technologies, captured under the umbrella term 'Web 2.0' that make it more interactive and dynamic, allowing entire applications that run in the user's browser. This makes it possible for us to provide an interactive web-based interface to theorem provers, and have web pages that are 'remixable': allowing users to reuse parts of the web, and formalizations in particular, in their own creations.

Combined, these two technologies can provide a platform that allows experts and non-experts to work on formal proofs: both by developing new proofs and by communicating about previous proof attempts. This thesis describes the integration of current web technologies with theorem prover tool chains. While we carried out this research, several other community-based web systems rose to prominence: the most well-known of these are StackOverflow, which provides a question-and-answer platform for programming, and GitHub, which allows programmers to easily share their code and accept changes by others. We focus on Wiki technologies here, because they allow low-level contributions by anyone, as opposed to the more specialized focus of StackOverflow and GitHub. However, the ideas used in these platforms do have a place in the theorem proving world, and have some influence on the designs in this thesis.

1.2 Tools of the Trade: Interactive Theorem Provers

In this thesis, we focus on the workflows of *interactive* theorem provers: proof assistants that not only verify proofs written in its language, but provide feedback while the author is writing the proof. Especially in Europe, the best-known current systems in this style are Coq[1] and Isabelle.[2] In this thesis, we do not focus on so-called 'batch mode' systems: proof checkers that allow a user to write an entire proof, and then verify it in a compilation step. If the proof is not correct, the author can correct any mistakes that the proof checker pointed out, and verify it again. While these systems, in particular the Mizar[3] system, are a part of the formal mathematical firmament, the batch-mode interaction style does not pose many new questions in a web-based setting: most of this thesis focuses on adding web-based interactivity to proof assistant documents. However, Chapter 4 focuses on a static rendition of formal proofs, and we do include Mizar in that discussion.

The interactive theorem provers all use the following model of interaction:

- The user writes a phrase, in the syntax of the theorem prover, that needs to be proven. This phrase, like all user-written commands, gets transcribed in what we call a *proof script* in this thesis. Across the field, this document can have different names, such as *proof document* or *proof text*. The choice for 'script' is fairly arbitrary, and based on our experience with the Coq theorem prover, that uses this name.
- The theorem prover responds by parsing the phrase and, if no syntactical errors are present, returning it as a *state*: a number of goals that need to be proven, combined with the hypotheses, including declared variables, that can be used to reach the goals.
- The user responds by writing a command, typically a *tactic* that is meant to *refine* the state. Tactics typically come in one of several forms.[4] While we choose Coq as a basis for this command classification, we believe that this list covers the commands that occur in proofs written in an arbitrary theorem prover, the difference being what types of commands are encouraged:

  Administrative tactics manage the proof state as a whole: these are, for example, tactics that introduce a variable to the hypotheses in order to prove a universal quantification in the goal.

  Proof flow tactics are used to break up a proof in sub steps, for example by asserting a sub goal that can be used as an hypothesis once proven.

  Application tactics apply a lemma or an assumption, or use an equivalence to rewrite the goal or hypotheses.

  Case analysis and induction correspond to the proof methods of the same names in mathematics, and break down the state based on the structure of a hypothesis or variable.

[1]http://coq.inria.fr
[2]http://isabelle.in.tum.de
[3]http://mizar.org
[4]Based loosely on the description of tactics in the Coq manual at http://coq.inria.fr/distrib/current/refman/Reference-Manual010.html

Computational tactics invoke the computation of a certain function. This is a feature exclusive to interactive systems, which can reduce a function application to some normal form.

Automation tactics apply some automatic procedure to determine the proof of a goal.

- Using the user-written command, the proof assistant refines the state accordingly, or gives an error if the command is not applicable to the current state.
- This process iterates: the user writes new commands based on the given state, which cause the theorem prover to compute a new state.
- Finally, the theorem prover reports that the goal is proven, and the user finishes the proof, normally by giving a command like 'Q.E.D.': in some proof assistants (notably, Coq) this causes a final verification of the proof by using the proof kernel.
- If the user determines the goal is unsolvable at any point in time, she can backtrack or abandon the proof entirely.

In Agora, we tap into this interactive process in several ways:

- We display a proof in Agora not just as the proof script, but allow a visitor to also see the states the proof assistant computed.
- We have Agora mediate between the proof assistant and the user, providing an authoring interface for formal proof.
- We use the proof assistant's knowledge of identifiers and their types to provide rich linking information. This is not limited to interactive proof assistants, and we also provide this functionality for Mizar.

The connection to the theorem provers is done without modifications to the theorem provers themselves: instead, we use the provers through the available (program) interfaces and supporting tools.

1.3 Collaboration or Communication

A traditional wiki, as mentioned before, invites visitors to collaborate by making editing any page easy. As also mentioned, part of the research in this thesis lowers the editing threshold for formal proof on the web, but another part focuses on the use case of communicating formal proofs to a reader. We choose to support communication for the following reasons:

- It is underdeveloped: while there has been some research on the readability of formal proofs, their power of communication has not received much attention.
- Success in this area opens up the field: communication is a way of exposing outsiders to a formal proof. While communication tools still need a skilled user to live up to its potential, without the tools, someone wanting to communicate a formal proof is left with crutches.
- Developing collaboration tools for formal mathematics requires more insight in the process of writing formal proofs, in particular to determine how to deal with changes made by collaborators that are physically separated.

- Regardless of the specifics of any collaboration workflows, there needs to be a significant speedup in the (re)verification step of proof assistants, to support a quick turnover on submissions by new users: this is not limited to just the Coq proof assistant, but has also been noted for Mizar [91], and a quick sampling of Isabelle's continuous integration system[5] shows that verification times for large parts of the Isabelle libraries also take several hours.

The reasons above make communication workflows a 'low-hanging' fruit, which we address in this thesis. During the development of the tools supporting collaboration, we have arrived at the implementation of a system that can potentially support collaboration on formal projects, although this requires more research and developmental effort.

1.4 Contributions and Outline of this Thesis

This thesis makes several contributions to the field of formal mathematics:

- identify and detail a number of workflows for *communicating* formal proofs and support these workflows with web-based tools (Chapters 3 and 4);
- identify advances in web technology that can support formal proofs and use these to support authors and explainers of formal proofs (Chapters 4, 5 and, to a lesser extent, Chapter 3);
- design and develop a platform for supporting the tools developed (the entire thesis, the actual design is described in Chapter 2);
- show the strength of the platform by including a non-trivial formal development and its description (Chapter 6).

The rest of this section gives a chapter-by-chapter overview of the thesis, including a pointer to the original publications, where necessary. Also where necessary, attribution to co-authors is given. Regardless of co-authorship in the source materials, the author of this thesis takes full responsibility for the content of these chapters.

Chapter 2: Architecture of Agora globally describes the architecture of Agora: a wiki platform that gathers sources of formal developments and supports descriptions of these developments on informal pages. In particular, it details the intended users of Agora, and the representation of its contents as *documents*. Beyond this technical description, it gives several plans for *validating* how well the user types fit actual users, and how suitable the workflows detailed in this thesis are for them. This validation is left for future work, because it did not fit in the scope of the originally proposed research, and can only be carried out with the knowledge obtained over the course of this research.

This chapter has not been published previously.

Chapter 3: Proviola describes the underlying data structure of Agora's documents: *proof movies* represent the dynamic nature of interactively developed proof by storing both the commands issued to the theorem prover and the corresponding responses. The responses in these movies can be dynamically obtained by a *Proviola*, a display technology integrated in Agora, but also available as a stand-alone tool. The chapter describes several methods of

[5]`http://isabelle.in.tum.de/testboard/Isabelle/summary`

enriching the data structure for web publication, leading to the final data type used in Agora's movies.

The Proviola idea and tooling has been described in three papers: "Proviola: a Tool for Proof Reanimation" [83] and "Narrating formal proof (work in progress)" [82] have been combined to become Chapter 3. "Dynamic proof pages" [87] speculates on the use of proof movies for wikis, but has not been used in the formation of the chapter. In the case of all papers, the co-authors have provided editorial comments and improvements.

Chapter 4: Point and Write describes the infrastructure Agora uses to support including formal mathematics in informal descriptions. This infrastructure consists of a *syntax* for including the entities within the narrative, and *semantic annotations* that can be used as 'signposts' for resolving the syntax into embeddable islands of formal mathematics. While the syntax is implemented as a part of Agora, the semantic annotations can be added to any web page, making it possible for Agora users to refer to formalizations outside Agora's own repositories.

This chapter is based on the paper "Point-and-Write: Documenting Formal Mathematics by Reference" [86], co-authored with Christoph Lange and Josef Urban. Lange is the developer of the OMDoc ontology used for semantic annotation, and also provided the description for that technology. Urban has implemented the semantic annotation for Mizar documents and described this part of the implementation.

Chapter 5: Authoring gives a technical overview of an authoring framework for Agora: it makes use of an off-the-shelf web editor for writing programming code, and attaches this editor to the underlying data structure, to obtain an asynchronous editor for formal proofs, that is no longer tied to the old read-eval-print model. The chapter provides an overview of the editing model, describes the protocols that are used to synchronize editor and server copies, and shows the power of the movie-based model by adding tools that compute on the movie representation of a formal proof.

This chapter is based on the paper "Proof in Context — Web Editing with Rich, Modeless Feedback" [81], written by the author of this thesis.

Chapter 6: Case Study uses Agora to display a chapter of the book describing a formal proof of the Kepler conjecture (the Flyspeck project). The chapter included in Agora was translated from the original LaTeX sources into Agora's informal language, and the formal proofs that the text refers to replaced by the inclusions developed in Chapter 4. For this chapter, the source code of Flyspeck is also supported in Agora, by creating glue code that translates HOL Light sources in the movies described in Chapter 3. Movie support for HOL Light sources also immediately opened up the web editor described in Chapter 5 for the Flyspeck code.

This chapter is based on the papers "Formal Mathematics on Display — A Wiki for Flyspeck" [85] and "Communicating Formal Proof — The Case of Flyspeck" [84]. Both papers were co-authored with Cezary Kaliszyk, Josef Urban and Herman Geuvers. Kaliszyk and Urban developed the proof advise service described in the chapter and described the service's technical details. Urban developed the script translating from LaTeX to the Creole syntax. Geuvers provided editorial comments.

Chapter 2

Architecture of Agora

2.1 Introduction

This chapter describes the design of Agora as a prototype 'Wiki for Formal Mathematics': the system implementation has occasional problems that make it less suitable for day-to-day use, but rather serves as a research prototype that contains the experiments described in the next chapters of this thesis. We do believe that the organizing principles described in this chapter are good enough to re-implement the system for actual usage and empirical validation of the workflows discussed here. The source code of the prototype implementation can be found at `https://bitbucket.org/Carst/agora`. A version of the system is running at `http://mws.cs.ru.nl/agora_cicm`.

This chapter will first describe the user categories that the system is designed for, including the use cases these user groups have, followed by an overview of how these usage patterns can be validated[1]. We then move on to the technological side of the design, giving a high-level overview of Agora's organization of sources into structured documents that are gathered into repositories.

2.2 Users

Because this research is mainly concerned with workflows for communicating formal mathematics, we first need to identify the potential users of Agora.

Instead of focussing on individual users, we will use user *stereotypes* here: each represents a class of users for the system, and a single user can inhabit multiple classes. Contrary to the traditional user *role* used in software engineering, the stereotype does not have fully specified use cases, instead, we give a high-level overview of scenarios these users might play. The role of user stereotypes is similar to that of a *stakeholder* in software engineering: it presents the goals and investments a particular party can have in the system. In the case of Agora, we do not focus on non-user stakeholders, but only give the three stereotypical end users. This means we also do not consider the users who set up and configure the system: their interactions with Agora are intended to be limited, and their workflows are of no interest in the communication of formal mathematics. They should be considered when implementing the system, as they have a role in deploying the system for end users.

[1]For reasons given in Section 2.3, the actual validation was not carried out. The Section should be read as 'future work'.

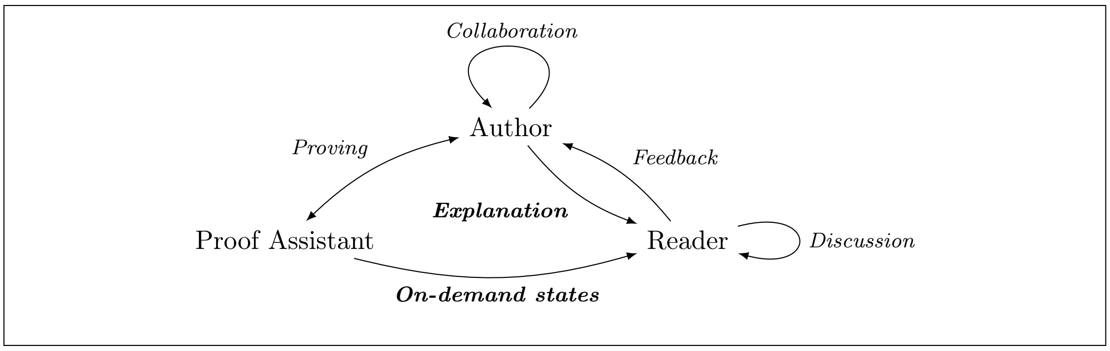


Figure 2.1: The communication triangle: normal text indicate agents, arrows, labelled in emphasis, indicate communication between agents.

A similar modelling tool from the interaction design community is the notion of a Persona [24], which is a fleshed out character that designers can use to illustrate how a user would approach the system. The approach we take here can be considered 'Persona-light': we consider the user roles as actual users of the systems, bundling a number of use cases and scenarios, but do not give any support for these roles beyond observations. Personas are well-documented descriptions based on empirical user research, while our basis of user requirements is more observational, based on the typical users on the proof assistant mailing lists and on the formalization projects developed in our research group.

We first consider three agents working with the content of Agora: the author of formal mathematics, the reader of the content and the proof assistant that verifies the information. The diagram in Figure 2.2 lists the possible ways these three agents communicate with each other, about a formal document.

The figure labels the arrows with the names that we use for the types of communication, with the main foci of Agora in bold face (for ease of reference, the face is mimicked in the legend below):

- the communication between theorem prover and author is interactive theorem *proving*. This type of communication is not changed in the presence of Agora: we investigate proving in the context of Agora in Chapter 5, but the scenario itself is not a new one;
- ***on-demand states*** are the major communication of the proof assistant to the reader: they allow the reader to independently discover a formal proof, without changing it, as explored in Chapter 3;
- ***explanation*** is the task of writing a human-understandable description of a proof that is to be consumed by the reader, something we describe in Chapter 4;
- *feedback* is the communication from the reader to the author. Agora's current implementation does not support feedback in the system (obviously, a reader could send an e-mail to the author to give feedback), but several texts, in particular in programming languages have been published on the Internet with support for reader comments, for example the text book "Programming Scala"[2];
- the self-loops on the reader and author points of the triangle represent collaboration between different users in these roles: when readers communicate about a formal proof,

[2]`http://ofps.oreilly.com/titles/9780596155957/`

$$\begin{tikzcd}
\text{Author} \arrow[loop above, "\textit{Collaboration}"] \arrow[r, leftrightarrow, "\textit{Collaboration}"] \arrow[d, leftrightarrow, "\textit{Proving}"'] & \text{Narrator} \arrow[loop above, "\textit{Collaboration}"] \arrow[d, bend right, "\textbf{\textit{Explanation}}"'] \\
\text{Proof Assistant} \arrow[r, "\textbf{\textit{On-demand states}}"'] & \text{Reader} \arrow[ul, "\textit{Feedback}"] \arrow[u, bend right, "\textit{Feedback}"'] \arrow[loop right, "\textit{Discussion}"]
\end{tikzcd}$$

Figure 2.2: The communication 'triangle' (now a square) revisited: explanation is now carried out by the narrator

this is *discussion*, intended mainly to clarify specifics of a document. For the author, the self-loop is actual *collaboration*: writing a proof together, possibly mediated by different tools.

There is no arrow returning from reader to proof assistant, because the when the reader starts instructing the proof assistant, it is no longer consider reading, but rather writing, so the reader would fulfill the author role. There might be some cases in which the reader queries the proof assistant for extra information such as giving the type of a function or searching for related lemmas, but these are not independent scenarios. Instead, they are one-off use cases that occur as part of the reader reading a proof: the reader queries the proof assistant to look up a certain definition or lemma. The author might also use this querying use case when writing a proof, looking for a lemma that helps in solving the current goal.

There also is no self-loop on the proof-assistant point in this diagram. While there are scenarios in which proof assistants can verify each other's proofs, we cannot imagine a scenario in which proof assistants collaborate on a proof, in particular not on their own accord. In many ways, the proof assistant is not an independent actor, as it is always driven by actions by the other players in the diagram.

We believe that the "explanation" arrow deserves extra merit: it has been under-represented in proof assistant research, but is an important scenario in science in general. To this end, we actively separate the author role into an *author*, who interacts with the proof assistant, and an *narrator*, who explains a formal proof to the readers. The collaboration arrow on the author remains the same: authors and narrators can collaborate amongst each other, something we saw in practice in the development of Homotopy Type Theory [74]. In practice, the author will usually also play the role of narrators, but there might be narrators who are only explaining formal mathematics, which justifies this separation. The separation of these two roles is displayed in Figure 2.2. The figure is an updated version of Figure 2.2, with the addition of an 'narrator' agent. Explanation is now the narrator's task, and the reader can give feedback to both author and narrator.

We will now discuss the three user stereotypes in detail, focussing on the arrows in the diagram.

2.2.1 Reader

The reader is the most important user of the system: without a reader, there would be no need for publishing the material within it. More specifically, every user of Agora is likely to be a reader, no matter what other stereotype they belong to: Agora is intended to serve as a repository of knowledge for people working on a formalization, and this means the users should be able to look up and read this knowledge.

The pre-existing knowledge of the readers in Agora varies greatly across separate readers: as evidenced by the conversations of users of proof assistants on their dedicated mailing lists [67, 47, 27], these users range from computer scientists to mathematicians, and from experts of a particular system (or multiple systems) to students just taking their first course in using a proof assistant. One of Agora's goals is to help bridge this experience gap, but it is not focused on displaying its content adaptively [19], but instead on giving authors and narrators the tools to manually tailor content to a specific audience, possibly reusing information that also appears in other displays.

Not all novice readers of the system will come to it with knowledge about formal mathematics, but one of Agora's goals is to support these novices in becoming authors of the system. This means that we do not only need to give passive information to the reader, but also allow exploration and interaction, so the reader can gradually master a formal system enough to contribute to the system's content.

The knowledge contained in Agora consists both of formalized mathematics (in the language of a proof assistant) and informal descriptions of this formal text. The reader comes at this knowledge armed with just a browser, intending to read through the information and get to know it. In particular, the reader has several ways of learning (how to use) Agora's content, which are explained in the next section.

Scenarios

For a reader, we need to support reading and browsing scenarios and experimentation with the formal entities in Agora, in order to allow them to *consume* the mathematics in the system, as constructed by authors (formal proof scripts) and narrators (informal narratives). Finally, the reader should be able to discuss the content with other users of the system.

Reading informal text Reading informal mathematical text can be supported by rendering the mathematical formulae similar to LaTeX, but also by including examples of formal source code. Beyond this, a web system should support exploration of the text by providing hyperlinks, which the reader can use to obtain more details about the concepts on a page, as well as related concepts. This exploration should not be limited to informal text (as is the case for Wikipedia), but should also support linking within formal texts and cross-linking between formal and informal mathematics.

Reading formal text To read formal text, the reader not only requires the proof scripts detailing the proofs, but also the contextual information the proof assistant computes based on the scripts. The reader should have access to the extra information on-demand, as argued and described in Chapter 3

Searching The current prototype of Agora does not incorporate search, but it is an important scenario. In it, the reader searches for information in different ways: in the informal text, similar to a database like Wikipedia; using semantic search tools, such as SPARQL [77], and using tools specific to formal mathematics, such as the content-based mathematical search engine, WHELP [7].

Experimenting Because the bulk of information in Agora is code intended to be interpreted by a proof assistant, the reader can be assisted in understanding the formal content by experimentation with the formal content. This experimentation can take several forms, such as trying out different alternatives for a particular formal definition, or directly using a definition to formalize a certain lemma. These experimentation opportunities can be divided broadly in two categories, tied to the reader's experience: experimentation can either occur in a sandbox, or in a guided exercise.

The sandbox is geared towards more experienced users: it should allow the reader to put a formal development through its paces, finding out in which situations certain formalizations and lemmas work, and how they can be applied to other situations. This means that the sandbox should not be tied to a specific development, but be available wherever the reader is in Agora. By this availability, we mean that the reader should be able to include specific formalizations and add to them by writing new code. On the other hand, the novice can be helped by providing guided exercises, which should indicate whether a formalization is (verifiably) correct, but also whether it matches a solution, to some extent. The latter verification could be done by an automatic system within Agora (as future work), but could also be done by a human teacher. What is needed for both of these cases is a way for the reader to start editing (pre-defined) islands of formal code directly where they appear in Agora: in the case of the sandbox this helps in starting an experiment that can then be kept around, in the case of an exercise it means filling in the exercise at the appropriate spot.

Discussion Finally, the reader should have opportunities to discuss the system's contents with other users, in order to ask for clarification and assistance or to suggest improvements. Such discussions should not be limited to just informal comments, but should ideally also allow a reader to include interpreted formal text. Such discussions should work like the comment systems, where readers can comment on sections, and also allow ad hoc conversations of the mailing lists specific to the proof assistants, or on web sites like Stack exchange [3].

2.2.2 Author

In this thesis, the *Author* stereotype is used to characterise the author of formal proofs. To carry out this task, the author uses an interactive theorem prover. Any user that writes formal text is an author, but most prominently, we focus on the scenarios of more experienced users of the system.

Such users are characterized by specialized workflows and tools beyond the proof assistants: typical tools include version control systems, as well as specific build configurations (libraries and settings) of the proof assistant. We do not have any quantifiable data on how authors use these tools. This implies that we can either guess how to support them based on our own experience, or first try and obtain any information. Obtaining this information is an opportunity for future research, potentially based on related research in software engineering: the more involved developments by experienced users have much in common with software engineering projects [37]. We opt for the first option, restricting the author scenarios to the simpler ones of publishing formal code and making minor, low-impact changes.

[3]`http://stackexchange.com`

Scenarios

In the scenarios for the author, we mainly consider the point in the lifecycle of the development when the author starts to use Agora.

Start of project In this scenario, the author uses Agora to start a new project. In this case, some of the reader scenarios also apply: the author browses the libraries in Agora in order to find developments which can be used as a basis, or obtain documentation for certain techniques. Beyond that, the author can start a new project directly in Agora, and use Agora as the development environment, or use Agora as a host for local development in the project, or a combination of both approaches. The first approach means using Agora for managing and writing formal code, and requires not only a good editor, but also support for versioning and team management. The second approach means using Agora as a web site from which the project's source code can be obtained, and through which contributors can upload their code.

After the project is started in Agora, the author can use the system either as a remote repository (similar to sites like Github[4]) that directly renders the proofs, or use Agora's editor to actually write a proof.

During project or after project When the author starts using Agora for a project which is already underway, it is possible that there are dependencies and idiosyncrasies in the author's workflow that are not supported by Agora. In this case, Agora should support adding new dependencies and tool configurations. These dependencies then support the author in working with the development in Agora, just like in the "Start of project" scenario. Having good support for dependencies does not amount to trivially uploading local libraries: it is possible that these libraries conflict with libraries already in Agora, either because they are named similarly or because they are a different version of the same library. This implies that we need proper configuration management in the system, possibly supported by version control systems.

When the author uses Agora after a formalization has been finished, the system should meet a project's dependencies as before. The difference in this instance is that the reason for uploading is different: it is meant either to use Agora as a repository for storing and disseminating the proof, or to use as a basis for future developments.

2.2.3 Narrator

There is a tension between the roles of author and reader: the author writes formal text, while the reader is aided by informal text written about the formalization. To relieve this tension, we explicitly introduce an *narrator* stereotype: this type of user bridges the gap between the reader and the author, by writing informal texts that contain formal islands. Because of the specialized knowledge an author possesses, users will typically have both an author and an narrator stereotype. However, because some readers are knowledgeable about the informal domain that is formalized, they can contribute here as well, giving explanations about the informal text. Finally, it is possible that the narrator is an entirely separate user, one who specializes in disseminating the formal work to a wider public.

It is the narrator role that is most suited to the editing workflows typical to Wikipedia (and other wikis). These workflows are intended to invite any reader to edit the content of the library, both by adding exposition and by cross-referencing articles, which is exactly the task of the narrator. In particular, there is a cognitive overhead for a reader to edit the formal

[4]`http://github.com`

content of Agora: before this can be done, the reader needs to learn how to write in the syntax of a proof assistant and the writing itself can make the entire formalization that builds upon a definition invalid or too slow to be verified. On the other hand, it is easy to write some informal text, making it less daunting for a reader to start with.

Scenarios

The narrator's scenarios publish informal texts about formal developments.

Writing and marking up informal text This first scenario is that of the traditional article author: the goal is not to create new formal content, but to write an informal description, a *narrative* about this formal content. Like traditional articles, narratives are written using a markup language and include formulae, figures and snippets of code. Additionally, this scenario also incorporates linking to other informal pages in the wiki. This scenario needs to be very accessible to new users: it should be easy to start editing, and the syntax used for markup should be close to the languages potential users, in particular mathematicians, use.

Cross-linking formal and informal content Besides linking the informal pages, the narrator should also be able to refer to sections of previously-written formal code. This linking should not be simple copy-and-pasting: since the formal development might change, this would lead to inconsistencies between the formal source text and the included portions. On the other hand, it should be possible to 'fix' a certain version of the text to be included, for example to describe a previous attempt.

Adding interactive elements Finally, the narrator should be able to add interactive elements, such as exercises and sandboxes. As described earlier, these elements are useful to the reader, and the narrator is capable of deciding where the elements are most effectively used in the narrative.

2.3 Validation

The user scenarios described before, and the workflows described in the next chapters are based on incidental observations and local experience. Because of the reasons listed below, we have not carried out a validation of how exact the scenarios are, and to what extent our workflows support the scenarios. We have obtained positive feedback on the Proviola technology described in Chapter 3 from users on the Coq-club mailing list. The case study in Chapter 6 shows that the tools can be applied to an actual development that contains both formal and informal text. This case study also generated positive feedback from the author of the original text, Tom Hales.

However, it is still desirable to execute a validation of the technology. Because the focus of this research project was to investigate the suitability of web technology for formal mathematics, and not to execute a use-driven design of an actual system. The following factors in particular prevented executing the studies from the beginning:

Time The most important reason is a lack of time: Agora has been developed in the context of the NWO MathWiki project, which had a duration of four years and was aimed at investigating the, mainly technology-driven, design of a Wiki for interactive theorem proving. This means that there was little time left for setting up empirical research that gathers user information, devises a solution, implements it and iterates, in order to see if the solution solves the problems the users were having.

User base For the field of formalized mathematics, the user base that can be drawn from for user studies is rather small, leading to a skewed coverage: there is a cluster of experts, all tied to their own systems and ecosystems, who might be interested in doing some user studies, but they would not encounter the same problems that beginners or intermediates might encounter. In particular, we do not have ready access to mathematicians that might be interested in using formal mathematics, meaning that we cannot focus on the users we would like to draw in. We could do experiments on master's level students, but the amount of time between the courses (our group is involved in teaching two proof assistant-assisted courses during a year), makes it difficult to iterate quickly. Beyond that, we would need to take care that studies on students give enough information, while not influencing the grading of the course.

Experience Finally, our research group does not have any real experience in running an empirical study, which can cause considerable setbacks and delays in solving problems.

The above reasons caused us to focus mainly on using proof assistant and web technology to produce new and improved artefacts. Clearly, it would strengthen the research and design of a system supporting formal mathematicians if we can quantify our observations as future work. To this end, we will need to reach out to experts in the field of usability. I will document some possible methods of validation on each of the different user stereotypes in the rest of this section.

Empirical data obtained about users should point out how users can interact with the pages in Agora. Mainly, we are interested in how readers search for information, either formal definitions for use in a formalizations or informal descriptions of techniques and developments, and how much they use page content, both the textual content and the interactive elements. In the case of authors, our main interest is their collaboration, and how their proofs evolve as a result of this collaboration. Finally, we need information on how narrators write documentation about proofs, how they *communicate* their proofs to readers.

There are several categories of users that all have their own data sources and who all interact differently with the system. We describe the following categories here: beginners with no domain knowledge, beginners with domain knowledge, and intermediates in using a proof assistant. Experts are considered as a source of author data. We mention narrators as a separate data source, split between the occasional Wikipedia-author and writers of academic articles.

Beginners with no domain knowledge Students are the exemplary beginners: a master's student in computer science (in the Netherlands) typically has no prior exposure to a proof assistant and its library, and normally has no domain knowledge about the subject formalized, in particular not in a course that teaches a subject *using* a proof assistant, such as Software Foundations [72].

To be able to gather data from students, we would need to set up a course that integrates using Agora, for example by publishing course notes as Agora pages and requesting students to use Agora as a facility for obtaining information necessary to complete course assignments. Measuring success can be done through student evaluation, similar to an evaluation of the Ask-Elle tutor for Haskell [34]. Additional information can be gathered by interviewing individual students and by evaluating (anonymzzed) server logs for queries submitted to the system and how the students browsed the pages: whether they used the interactive elements of page and what hyperlinks they followed. If we were to execute these experiments, we should take care that any problems within the system do not detract from the educational goals of a course: students should not be hindered in learning because a system is prototypical and, therefore prone to bugs.

Beginners with domain knowledge This group of users mainly consists of mathematicians, who are familiar with traditional, informal mathematics, and want to apply this knowledge in a formal system: either by formalizing existing theory, or by comparing an informal mathematical text to its formalizations.

It it difficult to obtain hard data from this group as the group is diversified and traditionally not very interested in formal mathematics. Their interest is increasing however, with significant formalizations of mathematics (such as described in Chapter 6) becoming more common. Most likely, individual interviews work best to obtain scenarios that mathematicians are interested in. This group is particularly interesting for obtaining information about the connection between informal and formal mathematics, and how to clarify these connections.

Intermediates Intermediates know how to use the proof assistants, and are typically working on a development. However, they are not experts, and are expected to search for specific patterns of using a proof assistant, or libraries that implement ideas similar to theirs.

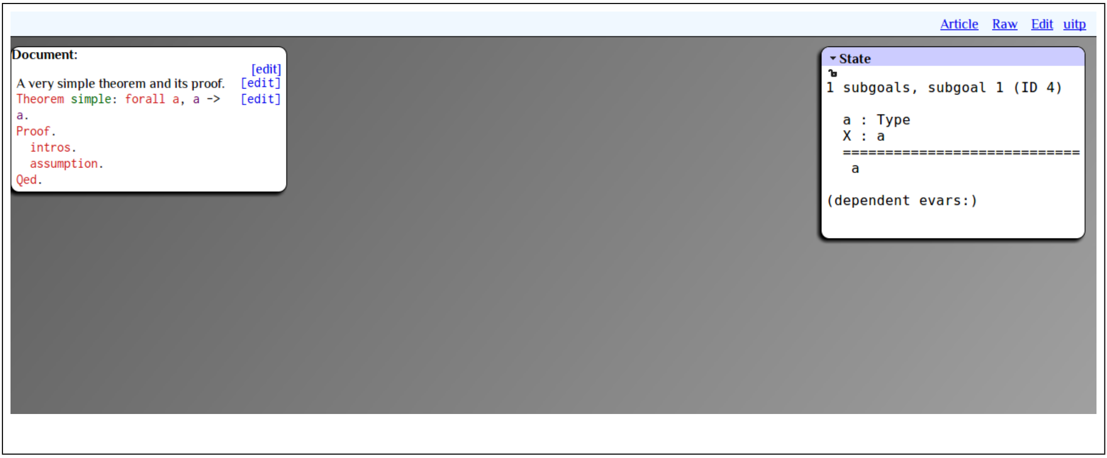
Because intermediate users flock to the prover mailing lists and other community information sources, we can obtain quantifiable data by mining the archives of these sources, in order to find common questions as well as the paths to resolution.

Authors We consider both intermediates and experts here, as their workflows are most likely to be similar. We do not have much information on how authors of formal mathematics write their proofs, beyond anecdotal evidence: the field is still small, and there are only a few major developments. There are parallels between software engineering and formal proof development, and we could benefit from the extensive research in that field. There are differences between the fields, however, so techniques from software engineering should be carefully evaluated. In particular, the following differences stand out:

- Once a proof is finished, it does not change much anymore, compared to a lot of software. It is possible that a proof gets rewritten for readability or to generalize parts for a future proof, but the requirements of the development remain fixed: provide a (verified) proof of a certain theorem. In software development it is possible, even likely, that a program evolves to meet new demands.
- The development of a formal proof has a fixed requirement from the start, creating the proof of a theorem, and all efforts can go into developing an efficient way of proving that theorem. In software, there is a set of requirements that need to be fulfilled by a system, which can be dropped or changed, depending on the budget of the developers.
- The teams involved in proof development are rather small: the team proving the Feit-Thomson theorem in Coq consists of at most 23 members[5], while software development teams can grow larger.

Beyond obvious user surveys and interviews, it is possible to determine how a formalization evolves by analysing the artifacts created during a development. In particular, the proof

[5]On June 11, 2013, `http://www.msr-inria.com/projects/mathematical-components/`, the 'Team' tab, listed eight current members and fifteen former members.



A very simple theorem and its proof.

```
Theorem simple: forall a, a ->
a.
Proof.
  intros.
  assumption.
Qed.
```

```
1 subgoals, subgoal 1 (ID 4)

  a : Type
  X : a
  ============================
   a

(dependent evars:)
```

Figure 2.3: A document in Agora, showing the state of a proof

scripts encode semantically rich notions, which could be analysed by adopting techniques developed in the *Mining Software Repositories* research community[6]. Since most developments are administrated using a version control system, we believe we can analyse the history provided by such systems in order to obtain information about the way a development evolves under collaboration. This information is important in supporting authors that want to collaborate using Agora.

Narrators While some narrators in Agora might be similar to Wikipedia's narrators, we do note that Wikipedia has a larger contributor base than a formal system can ever hope to achieve. This means that while some lessons can be learned by looking at Wikipedia's workflows, they need to be tested on how they would work with less users.

The other source of data is authors of academic articles. Information from these sources is probably best gathered on an individual basis, searching for the common denominator: each narrator is different, but they all have the same goals and probably similar ways of achieving those goals. There has been some research on the writing of scientific literature, in particular the mining of the arXiv data by Cormode et al. [29]

2.4 Documents

Agora gathers the artefacts of formal proof, in particular proof scripts and descriptions of proofs, and allows the users to interact with them by writing and reading. These activities follow the scenarios described before. The user interacts with these documents through a web interface, which is shown (for a particular document) in Figure 2.4.

This figure shows how several tools, working on the underlying script, come together to provide a single interface to the user:

- Most noticeably, the document part of the proof is rendered as HTML, this rendering was done by the rendering tool of the proof assistant (Coq, in this case), to make use of the semantical information the tool has.

[6] `http://msrconf.org`

- Secondly, the proof state shown corresponds to the command pointed at. This state was computed by the interactive toplevel of the proof assistant.
- The third feature of this page is that the content is verified by the batch-mode checker of the proof assistant.

The computations used for these features are illustrated here for Coq, but can be different for another proof assistant. For example, Mizar generates the documentation as part of the verification step, and has no interactive toplevel. Despite the different tools involved, the interface should be uniform across proof assistants, providing the same functionality (where possible) at the same locations, no matter the underlying system. This means that we need a way of dispatching the functionality, which is offered through the web interface, to the underlying tools and proof scripts. To keep this dispatch as clear as possible, we abstract from the separate files and tools (and tool configurations) into a single data structure, the document.

The document is based on the proof movie data structure described in more detail in Chapter 3. That chapter focuses on the content and construction of movies. In this chapter, we focus on how the movie acts as a generic data structure for providing the functionality required for Agora. For this discussion it is only necessary to know the following of the data structure:

- It consists of a set of *frames*: extensible containers that represent individual commands in the syntax of the proof assistant. The extensibility comes from the possibility to add extra information to these commands, such as responses from the interactive toplevel or markup.
- Structure on the frames is provided through a tree of *scenes*. Each scene can have any number of children, which can be scenes or (pointers to) frames.

In the present chapter, we extend these notions with:

- Movie generation for markup languages, so that no matter what files are underlying a document, the same structure can be assumed.
- A description of scenes that are used to support the scenarios described in Section 2.2.
- A mapping of the movie data structure to high-level functionality.

2.4.1 Markup languages

Agora offers the Creole [76] syntax with a slight modification explained in Chapter 4 to narrators writing informal descriptions in Agora. Creole is a language that is similar to the markup language provided by Wikipedia, offering:

- inline text markup, such as italicization and bold face;
- block-structure syntax, which allows the author to write paragraphs and then combine them into sections by providing headers;
- a simple hyperlinking syntax, for links to other pages both within the Wiki and outside it; and
- an inclusion syntax for formal entities (as described in Chapter 4).

For formal languages, we split a script into frames based on the computational unit of the proof assistant, a command. In the case of markup, we do not have a notion of command, and we define it to be the structural elements defined by the Creole syntax. Each frame contains both the defining syntax and the HTML markup generated from it. The structural elements give us a natural organization of a markup text into scenes: each heading with the paragraphs following it forms a scene, with sub-scenes defined by sub-headings (and the paragraphs following that).

Inclusions are treated as belonging to a special scene type, described below.

Despite having used a fairly simple markup language, the characteristics described above also hold for a complex language like LaTeX, allowing Agora to include such a language by having a parser generate the same tree. The main hurdle that is keeping this inclusion back is defining a map from LaTeX syntax to HTML and to the scene structure. To aid in this definition, a tool such as LaTeXML [79] can be used to transform the raw sources into a structured XML tree.

2.4.2 Scene types

There are several scene types that represent the content of both formal and informal texts. These scenes have the added usability of supporting the scenarios of Section 2.2.

Because a scene can contain sub-scenes, the type of a scene determines how Agora treats its sub-scenes, as described in Section 2.4.3. For simplicity, we describe scenes containing only frames in this section.

Code The code scene is used to represent formal code. It is used in almost all scenarios. In particular, it can be pretty-printed for a reader of formal text, indexed for searching formal text, and used for communicating a proof assistant. The sandbox described for the reader's "experimenting" scenario can also be implemented as a code scene.

Because Agora supports multiple proof assistants, a code scene should indicate in which language its contents are written. Furthermore, a scene might depend on another scene: the proof assistant should first process dependencies before attempting to process the dependent scene. In our current implementation, Agora takes scenes to be linearly dependent, with the frames capable of containing a more fine-grained dependency graph, instead of (also) requiring a dependency relation on the scene level.

Code scenes can contain sub-scenes that specify the semantics of formal code, see Chapter 4 for more details on this concept.

Documentation Documentation scenes provide plain, human-readable documentation, supported by markup. This documentation can, for example be generated from a dedicated markup language like Creole [76], or from a documentation language embedded in the formal code, such as Coq's Coqdoc. Documentation scenes are used to communicate informal text, and are used as sub-scenes for several of the other scenarios.

Inclusion Inclusion is a concept that is strongly linked to Agora's goal of writing *about* a formal text. To keep the inclusions linked to their target, we add *inclusion scenes* to a movie, that refer to the target scenes. These inclusions are replaced by the actual scene when necessary, for example at render time. The targets of inclusions can be any type of scene.

In a future implementation of Agora, this scene can be extended with a "version" argument, which allows inclusion of a specific version of a scene. This provides a better internal consistency for presented document, at the cost of not showing the most up-to-date version of a scene that is available in the system. This idea has been discussed by Kohlhase and Kohlhase as their concept of *versioned links* [57].

Alternatives The alternative scenes are not part of the current design and implementation of Agora, but can be added to support the other scenarios in Figure 2.2.

Alternatives provide different ways of processing a scene, either informally or formally. Possible applications include giving a different explanation depending on the experience of the reader (for example, a mathematician benefits from a different explanation than a computer scientist), while keeping the full formalization for the proof assistant, or providing a solution to a formalization exercise in a textbook that is only shown to a teacher.

Alternative scenes are used in the informal communication of a proof, but can also be used for exploring different formalizations of the same concept, either by the author, or by readers participating in discussion or feedback: by giving an alternative, they automatically set up the context in which the content is interpreted.

A special kind of alternative is a comment scene, which also refers to a scene to set up a context, and then gives some commentary about that scene.

2.4.3 Providing functionality

Agora's provides functionality by allowing the user to manipulate the scene structure. These manipulations are exposed as URLs, that are used by the Web interface either as plain hyperlinks for navigation, or in JavaScript calls for interactive elements. The following functions are provided.

Rendering Rendering takes the scene structure of a movie and transforms it into an HTML page. To mimic the structure of the scenes on the page, we use the basic structure element of HTML, the `div`: each scene gets (recursively) mapped to a `div`, that contains the rendering of its children. To render a frame, the markup contained in the frame is shown. Additionally, the rendering step creates a table of content for the page, by gathering all the scenes that have a title specified.

To give users access to the editing workflows, each scene is decorated with an 'edit' link, which allows the user to start editing any scene, including its children. The dynamic display provided by the Proviola is only added to rendered code scenes, since documentation scenes do not contain code interpretable by a proof assistant.

Editing Editing in Agora is done on a per-scene basis: an author or narrator selects a scene to edit, and that scene is loaded, including its children, into the system's editing tool. Chapter 5 explains Agora's interactive editing model in greater detail. Here, we focus on the transformation of a scene into editable form.

To make a scene editable, all the frames that descend from it should be transformed into plain commands, without any markup. Since a frame stores the commands next to markup, this is just a simple tree traversal that projects out the 'command' part of each frame it encounters. However, we have to consider the fact that there are different types of scenes and they form a tree: this means that a documentation scene can contain code scenes and vice versa. We need to represent these embeddings during the translation of scenes to code.

The most straightforward approach to the translation is translating each frame in the scene directly into the command it contains, and this is the approach that the current implementation of Agora takes. However, this has some problems that need to be resolved for better support of the narrator and author roles:

```
Lorem ipsum **dolor** sit amet [link]

{{{ .coq
Lemma foo: forall x, x -> x.
Proof.
  trivial.
Qed.
}}}
```

Figure 2.4: Informal text embedding formal source

```
(** Lorem ipsum _dolor_ sit amet [link] **)
Lemma foo: forall x, x → x.
Proof.
  trivial.
Qed.
```

Figure 2.5: Formal text embedding informal documentation

- The narrator can accidentally modify the formal code, while only wanting to change the documentation for that code.
- The narrator might not be used to the documentation markup for a certain system, but still want to edit documentation for it.
- To support the author, the code scenes should be supported with interactive feedback, which overloads the narrator. On the other hand, the author is not concerned with previews of markup, which can be useful for an narrator.

To support both roles, we suggest a design which translates the children of a scene in a context-aware manner: when a user edits a documentation scene, any descendant code scenes are clearly marked (such as in the source code example of Figure 2.4) and when a user edits a code scene, any underlying documentation scenes get transformed into comments (such as in Figure 2.5, which uses the Coq syntax). These figures show an additional feature, that is not yet implemented in Agora: the translation to an editable text is not only aware of the different scenes, but can also translate between different markup languages. To support this, we need more information on the differences between markup languages (as not all markup languages support the same set of features) and a model that stores the markup in a language-agnostic way, inside the frames. Then, the system can translate to markup or editable code on the fly, based on the target language.

Next to the different ways of generating editable text based on code and documentation scenes, we also need to decide how to handle reference scenes. Currently, Agora translates references to the referring syntax. The alternative is to allow the users to change a referred scene directly through the reference. This is an interesting idea, as it allows the user to change a scene where they see it. The main problem with this approach is that a scene that is displayed by reference might be used by a completely different document. Handling this requires better support of verification than is currently in Agora.

Verification Verification comprises the extraction of formal text of a document and running an appropriately configured theorem prover on it. The result is cached on the document and displayed to the user. This result is typically that the content is correct or that there is an error (or multiple errors) in the text. Most theorem provers return the locations of errors, together with some description of what went wrong, which is also stored in the document's data structure. In Agora, verification is done after editing, meaning we have all code scenes flattened, then edited, then verified. After verification, the scenes are reconstructed.

This verification process is oversimplified: it only concerns the *internal* consistency of a document: that its content is correct, with respect to other document it imports. What it does not verify is the *external* consistency of a document: the correctness of documents relying on this document: it is trivial to make a document internally consistent by removing its content, but this makes other documents that include the changed document no longer consistent. The verification of documents is not necessary for informal Wikis such as Wikipedia: in those systems, there is no formal notion of (any) consistency. For these systems, there is a notion of links between pages, which are informally checked by contributors to the Wiki: when the target of a hyperlink is not found, the link is rendered in a recognizable way (typically in red). When a reader sees such a link, they can try to repair the problem: either by changing the link to point to an existing page, or by writing the missing page.

The interplay between documents means that we do not only need to verify internal consistency, but also external consistency. We will describe the verification in Section 2.5, which describes the organization of documents.

2.5 Repositories

While documents provide a convenient abstraction from files and interactions with those files, formal developments are rarely constrained to a single file: typically, such a development consists of multiple files that together represent a theory, where later files (normally) prove the main result, building on top of definitions and lemmas defined in earlier files. Additionally, the larger formalizations are worked on by not just a single author, but by multiple users in the author and narrator stereotypes.

This means that we need to organize the documents in such a way that the organization supports multiple users and is aware of the dependencies between documents. These dependencies are used to support users in their interactions with documents and to guide the theorem provers in the verification task, especially to maintain the *external* consistency.

To this end, documents are organized into repositories: a repository collects a number of files and manages instantiation of the documents from the underlying files. Additionally, the repository provides an interface to the file system for updates and provides access to configured tools whenever a document action requires them.

The term "repository" is borrowed from version control systems and the central code hubs that grew around them (*e.g.* GitHub, Bitbucket, or SourceForge), and this is what inspired the organization of the documents in repositories. Actual version control is roughly implemented in Agora and left for future work. This integration would make it easier to tie in with the author's workflows. We also believe that integrating community features based on the source code hubs is useful for involving readers and narrators in writing formal code.

2.5.1 Version control

Version control systems allow users to manage files in collaboration: they submit a new version of a file after editing, and the system supports reconciling multiple edits happening at (nearly) the same time, through the process of *merging*: after a merge, the file in the versioning system represents the changes both users made. Should the changes be in conflict (for example, touch the same lines in a file), the user should do manual conflict resolution. For centralized version control systems such as Subversion and CVS, there is always one recent version of the repository, stored on a centralized server. Each user can obtain that version, and changes need to be moved back to the server. Distributed version control systems, such as Git and Mercurial, on the other hand, do not have a central, up-to-date version. Instead, each user maintains a *branch* of changes compared to a previous version, and this branch can be distributed to other users, who then merge the changes into their own branches.

This branch-and-merge model of version control can help us in managing repositories of documents:

- Any user can freely experiment with changes to scenes (or entire documents) by creating a branch. If a branch turns out well, it can be integrated in Agora's version of the document. This experimentation is not limited to the source documents defining a scene, but also to locations where a scene is referenced: a branch can apply only to the occurrence of a scene within an informal text, or to the source document.

- Verification can be done on an entire branch of related changes: the system can verify the external consistency of documents with respect to this branch, and accept it as correct or point out problem spots.

This is currently not implemented in Agora, because it requires a way to represent branches within the system, so that they can be applied to the scene structure, and because it needs a well-designed interface through which users can manage branches from within Agora: there are some tools that help in management of branches that are part of version control systems, but it is not obvious how to integrate them in Agora.

2.5.2 Community features

There are several portals for open source development tied to version control systems, both centralized and decentralized. Beyond offering hosting of (branches of) a repository, these web sites also offer their users tools for inspecting and commenting on different changes and branches. The interfaces offered can serve as an inspiration for future work on Agora's repository management.

Chapter 3

Proviola

3.1 Introduction

In Chapter 2, we have described the generic organization of Agora's data in a collection of documents. In particular, we described documents as a data structure with functionality defined on it. We did not go into detail on the specifics of this data structure, or how the rendering functionality is implemented. This is the goal of this chapter, which is based on two previous papers [83, 82].

The **Proviola** data structure (a proof **movie**) has first been described [83] as the memoization of a proof assistant session, captured by submitting a proof script to the prover and recording the responses. The movie does not just keep commands and responses, but can be extended with arbitrary data. This allows Agora to use the movie not just to store proof scripts, but also informal documentation. The data is stored in **frames**, with a structure overlaid on it through the use of a tree structure comprised on **scenes**. The resulting data structure can easily be rendered as an interactive HTML page, making it ideal to use in Agora. However, the technology is also usable stand alone, as a tool that transforms a proof script into a movie and then into a web page.

We will first motivate the main usage of the Proviola, making formal proofs available to inexperienced readers, and then introduce the data structure in more detail, followed by the construction of the movie through parsing both plain proof scripts and marked up HTML. We finally describe the generation of Agora's HTML pages out of the data structure, as well as present an old idea on allowing narrators to write a narrative as a scene structure, a tool that inspired the syntax described in Chapter 4. All this is illustrated by a case study on a transformation of the Software Foundation course notes [72].

3.2 Motivation

It can be difficult for a reader to understand a formal proof: in preparing a proof for a proof assistant, a proof needs to be transformed to a specific syntax and style. This style typically requires more administration than an informal proof, for example by spelling out analogous cases and by making small, computer-understandable (but *not* human-understandable) steps, a growth factor Freek Wiedijk calls the De Bruijn factor [101]. Additionally proofs for a procedural proof assistant, like Coq or HOL Light, are written interactively: the author writes commands that cause the proof assistant to refine and destruct a proof state. The transcript of these commands is stored in a proof script, which is typically distributed as 'the' formal proof. Because of these reasons, however, the proof script is typically not understandable as a stand-alone document: the reader needs to be able to work with a proof assistant that can interpret the script and conjure up the proof states. This leads to a cognitive overhead in the form of at least:

1. It is necessary to become (at least somewhat) familiar with the technical details of the proof assistant, since the user needs to install and configure it before use;

2. The user needs to understand the tools the proof assistant gives her, in terms of libraries and commands, and how to use them to achieve a formalization of the proof. Not only is this important for the author, who needs this knowledge in writing, but also for the reader, who needs to be able to find out what roles these libraries play in the proof under investigation;

3. The user needs to know the proof and its implicit assumptions in far greater detail than required to communicate the main ideas to another person: the proof assistant requires that the proof is explained in greater detail than a human requires, with a justification even for 'trivial' steps.

Much of the effort in interaction design for proof assistant s has focused on the second and third of these issues from the point of view of defining a language of suitable basic proof steps, augmented with automation layers which are either fully programmable, or else encapsulate well-defined larger-scale proof steps, together with an editing model of how to soundly maintain a partially completed proof.

That is to say, the basic proof assistant use case of "writing a proof" has received most attention, while those of "reading a proof" (written by someone else) or "browsing a library" rather less so: the narrative or explanatory possibilities afforded by a formal proof text have been largely overlooked in favour of (variously prettily rendered) static digests of named definitions and theorem statements. These are necessary prerequisites for these use cases, but hardly sufficient for gaining insight into how such proofs 'work' (or even: how partial proof attempts may *fail*, as when trying to discuss such examples on a mailing list).

In each use case, however, there still remain the computational and cognitive bottlenecks arising from the first and second problems. For example, as illustrated by Kaliszyk [50], before being able to formalize a theorem in Isabelle, a user needs to install the system, comprising the program itself, an HOL heap and a version of PolyML. While the process for Isabelle has improved in the years since Kaliszyk mentioned this, there are still similar issues with other proof assistants, such as Coq, which needs to be compiled on Linux systems and is difficult to configure on Mac OS X. Having installed the proof assistant, the user is then left with understanding it: digging through a tutorial or manual, and finding out what contributions and libraries are necessary for the formalization of the proof.

Previous work at Nijmegen has partially addressed the first issue. By providing a generic web-interface for proof assistant s, the ProofWeb system [50] removes the computational load on users by uncoupling interaction with a proof assistant via a dedicated webserver. This relieves the user from the installation problem: she can just visit a website and use a web-interface to access the proof assistant. However, when trying to understand a (part of a) formal proof script, it is often necessary to see how the proof state changes through execution of a specific tactic. This requires first to bring the proof assistant into that specific state — finding and loading the required libraries and files — a significant overhead, not addressed by ProofWeb.

Similarly, when explaining a tactic or a part of a proof script, with the current technology, a proof author has to publish the proof script, sometimes with an explanation of the output the proof assistant returns to her. This is not very satisfactory and often not informative enough, especially when the publication of the script is intended to show the intricacies of the proof: if the reader is uninitiated in the specifics of the proof assistant, she might not understand the proof script.

In this chapter, we consider the following scenario in particular: the author wishes to communicate a script to a reader, who might not have prior experience with a proof assistant and is definitely not an expert in using the system. This restriction on reader expertise means that to interpret the script, it should be embedded in a document satisfying at least one of the following properties:

- it is enriched with a high-level narrative, explaining why certain decisions (in design, representation, tactic invocation, *etc.*) were taken and what their effect is;
- in the case of a tactic-based language, the proof script can be resubmitted to the proof assistant, so the reader can evaluate the effects of each tactic on the proof state; or
- the proof language in which the document is written mimics closely the vernacular of informal mathematics.

In this chapter we further consider specific instances of author, reader and proof assistant. These instances are chosen to represent artefacts that are already available to us, and allow us to consider how to improve the artefacts and workflows.

Author The author is writing a coursebook for use in a computer science curriculum. The book does not necessarily have to teach the use of a proof assistant, but can present a formal model of (a slice of) computer science that is verified by the proof assistant. This particular content means that the proof assistant technology is not on the foreground: instead of focusing on specific constraints imposed on proof style, the formal proofs in the book are meant to illustrate and provide more certainty.

Reader The reader then becomes the prime consumer of a coursebook: a student taking the course. We assume the student has no prior experience with the proof assistant used to write the coursebook. This restriction means that a student has much to gain by the dynamics of the Proviola implementation.

proof assistant For concrete examples and tools, we focus on the Coq system and its associated toolset [88]: this choice is motivated by our local expertise, and the existence of at least two coursebooks written in the form of a Coq script. These books are "Software Foundations" by Pierce et al. [72] and "Certified Programming with Dependent Types" by Chlipala [22]. Despite this choice, we believe that the techniques illustrated here are also applicable to other proof assistant s, especially tactic-based ones. In particular, we have verified this idea by implementing a prototype tool for the Isabelle proof assistant[1] and tooling for HOL Light (see Chapter 6 for details).

Choosing a coursebook as a concrete proof document allows us to make some assumptions about the content of such a document:

[1]Available through `https://bitbucket.org/Carst/isabelle_proviola`. Since this tool was developed using the API to Isabelle 2012 and has not been brought up-to-date since then, we do not discuss it in detail here.

- The non-formal content of the document is structured in chapters, sections, subsections and paragraphs.
- The formal content of the document is the underlying 'spine' of the document, subservient to the total narrative of the book. At some points, the commands might be brought to the foreground to be explained or to serve as an example or exercise, but the text explaining it is just as important as the proof script.
- To improve a student's understanding, the coursebook contains exercises. We assume these exercises consist of proofs or definitions that have holes in them, to be filled out by the reader.

A coursebook created as a Coq script generally exists in two different forms:

1. A rendered version of the document, in which the narrative is displayed together with the formal content. The rendering is meant to reinforce the reader's assimilation of the text, using bullet points, emphasis and other markup.
2. The script itself, loaded in an interface to the proof assistant such as CoqIDE or Proof-General [10]. This gives an interactive view of the document, allowing the student to step through the tactics and see their effects, as well as fill in holes in exercises. The version displayed in the interface does not have the markup of the rendered version.

These two modes of display correspond to the first two ways of assisting a reader in understanding a proof document: describing a proof using a high-level narrative and reviewing the proof script dynamically, by loading it in a proof assistant and stepping through the tactics.

Switching between a rendering of a document and the script requires a reader to switch contexts between the renderer and the proof assistant: to our knowledge, no interface to a proof assistant actually renders the documentation of a proof document in a nice way, and the rendering does not incorporate the proof assistant output based on reader focus. The tmEgg tool [66] is an attempt to do this, but requires the document to be written and viewed using the TEXmacs editor. Additionally, installing and configuring a proof assistant requires effort on the part of the reader, effort that we have lightened by integrating script and output in the form of a proof movie.

3.3 Communicating a Formal Proof

This section describes the existing use cases of communicating formal proofs, from the author to the reader. To this end, we first describe the two use cases of writing and reading a proof, executed by the author and the reader, respectively. These use cases describe the current usage of proof assistant technology, but serve as a basis for the new use case of the Proviola: that of 'watching' a proof, described in Section 3.4.1. The figures accompanying the use case descriptions are based on UML, but use the following conventions:

- A stick figure represent a user role (proof author and proof reader).
- A 'package' represents a tool/program instance (here: the proof assistant s).
- The cloud represents the Internet.
- A folded page is a file (here: the proof script).
- Arrows represent data flow; a double arrow implies interaction between two parts.

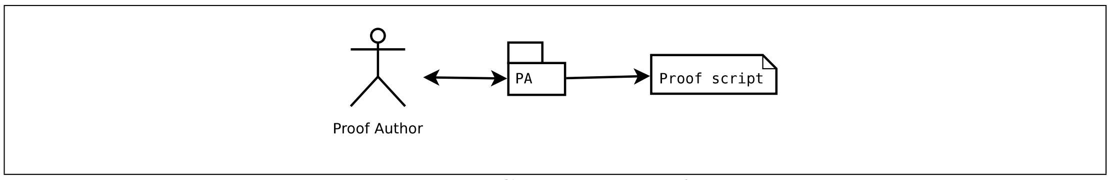


Figure 3.1: Creating a proof script

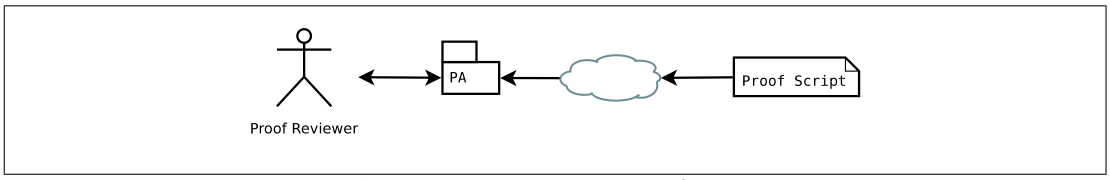


Figure 3.2: Reading a proof script

Writing a Proof To create a proof, an author writes commands to be interpreted by the proof assistant. In response, the state of a proof changes by decomposing the theorem to be proved, generating new proof obligations or discharging goals as proven. A proof script stores a transcript of the commands issued.

In Figure 3.1, we display the traditional implementation of this use case, in which the proof assistant is locally installed, and creates a local copy of the proof script.

Note that in this use case, the commands in the proof script are given by the author in response to proof assistant output, so the script is only one side of the conversation that generated the proof. Typically, however, it is this side of conversation that is published as a formal proof, with the tacit assumption that the reader has his own proof assistant for replaying the conversation.

Reading a Proof To read a proof, the reader obtains a (copy of a) proof script, possibly via the Internet. Subsequently, he can load it in his copy of the proof assistant, and 'replay' the proof: many proof assistant interfaces have a notion of stepping through a script, by sending commands one-by-one to the proof assistant or by undoing the last command sent. Because the proof assistant does not know that the commands it receives are extracted from a script, it responds to commands in the same way as in the creation use case: by sending the new proof state.

The interaction between reader and proof assistant in this use case is illustrated in Figure 3.2.

Discussion The communication style describe in the previous two use cases has two problems:

1. If only a small part of the script is relevant, it might still be necessary to send the entire script to the reader: the parts of interest might require definitions defined previously in a script, or might use lemmas proved earlier.

2. Before a proof can be reviewed, the reader needs to install (ideally the same version of) the proof assistant the author used, or be so familiar with its technology to interpret the script mentally[2]. Especially when the script is used to communicate a proof to a reader who is not part of the proof assistant community, this can be a large handicap.

[2]For a proof assistant that uses a procedural proof style (tactics), it is hopeless to try to interpret a proof script purely mentally, without seeing it executed.

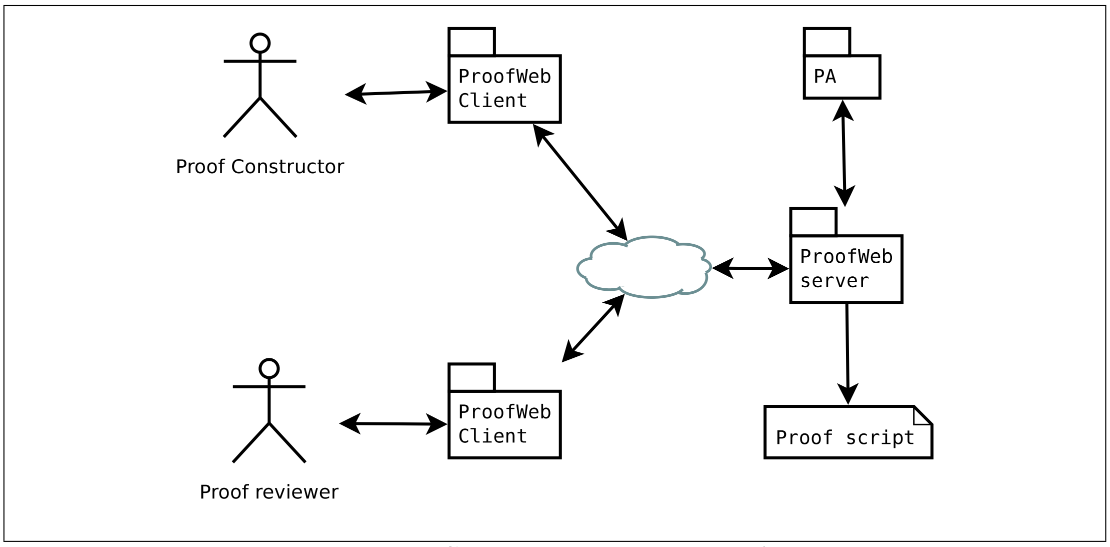


Figure 3.3: Communication using ProofWeb

A possible solution to the first problem, frequently exercised on the Coq-club mailing list [27], is to simplify a proof script to a minimal example, which focuses on the problem in the script, and the definitions directly necessary to obtain this problem. But such simplification might be too drastic, abstracting away crucial details. Furthermore, abstraction is not an option when the main purpose of the communication is not to point out a problem, but to display a (partial) formalization of a proof to an outsider, because it is necessary to stay close to the vocabulary and methods of the target audience.

The ProofWeb system developed by Kaliszyk [49] places the proof assistant on a central server that is accessible through an AJAX-based web application. This means that to review a proof, a reader needs only a web browser and the proof script, which could be hosted on the same server as the proof assistant, an architecture shown in Figure 3.3. The architecture effectively puts proof assistant behind the Internet (from a user's perspective), but does not change the reading and writing scenarios in any other way.

ProofWeb is an Internet-mediated realisation of the 'creation' use case, which mitigates the second problem of having to install and configure a proof assistant, but does not yet allow partial communication of the proof to a reader. It does not provide fast access to an arbitrary proof state: to obtain the state after a given command, all preceding ones need to be resent to the proof assistant for reprocessing.

3.4 Proof Movies

To solve the problem of computational overhead involved in obtaining an arbitrary proof state, we have enriched the proof script data structure. In the new data structure, which we call the **proof movie**, we record the commands sent to the proof assistant coupled together with the response of the proof assistant to each command. Such a pair of a command and a response is a **frame**. The exact Document Type Definition for the movie data structure can be found at `http://mws.cs.ru.nl/proviola/movies/film.dtd`.

The proof movie is designed to be *self-contained* and *generic*:

Self-contained Making the movie self-contained means that a proof reader only needs a movie

```
<frame frameNumber="2">
  <command>
    intros A x.
  </command>
  <response>
    1 subgoal

    A : Type
    x : A
    ============================
    A
  </response>
</frame>
```

Figure 3.4: An example frame of a Coq movie

and a tool capable of displaying it to replay the proof: no other tools are necessary for this. Aside from this, the frames themselves contain the exact state of the proof at the point represented, meaning that it is possible for an author to publish a proof partially, omitting or reordering frames before publication.

Generic By making the movie generic, creation and display do not depend on a specific proof assistant or proof assistant version: this does require that one can specify a transformation from the proof assistant's interaction model generating discrete frames.

Within Agora, movies are kept as the underlying data structure for documents. For persistence, Agora stores the movie in the eXtensible Markup Language (XML), which is also the output format of the stand alone Proviola. An example frame in this implementation can be found in Figure 3.4. This example contains the notion of a 'frame number': a sequence number identifying the order in which the commands were submitted to the proof assistant.

We do not consider the movie to be a complete replacement for a script. Instead, it is a container of a part of the script, together with the output of the proof assistant. This output does not need to be correct, but this does not interfere with our intention of the movie: we see a movie as an explanation of a proof, not as checked proof script *per se*. If a movie contains a complete script, the concatenation of all the command segments of all frames in the movie reproduces the script, which can then be (re-)checked by a proof assistant.

The movie introduces a new use case, *creating a movie*, which we describe in Section 3.4.2. Having the movie also changes the use case of reviewing a proof script, and we describe this modified use case first, in Section 3.4.1.

3.4.1 Watching a Movie

When the reader has obtained a proof movie, he wants to access the data within it to review the proof, much like when he obtained a proof script. In effect, the "reading a script" use case described in Section 3.3 and illustrated in Figure 3.2 has not changed, only the data structure supporting it has changed.

We call the system used for displaying a movie a **Proviola**. Just as in film-making, where an editor uses a Moviola to review a film while editing, and moreover quickly fast-forward and rewind the movie to see individual shots, we wish to achieve similar access speed and

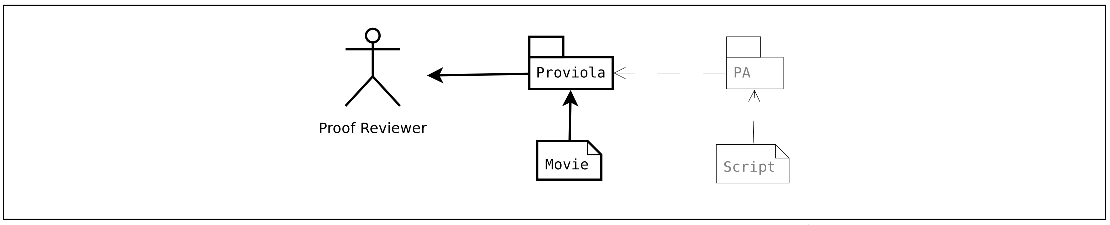


Figure 3.5: Watching a movie: Proviola and movie proxy proof assistant behaviour

portability. Indeed, our Proviola is not a separate tool: rather, we realize it through the use of HTML and very simple JavaScript code.

Reading a proof script, revisited: watching the movie The movie is self-contained, and can be distributed like a script. Unlike a script, the contents of a movie can be inspected without any external tools except a web browser: the movie can be located anywhere, and inspected from this location. In particular, a proof assistant is not required to compute the proof's state and the movie can be watched offline, at any time. After the reader loads a movie in the Proviola, he wants to step through the proof much like when a proof assistant was loaded with a script: by indicating for which command he wants to see the response.

As illustrated in Figure 3.5, the Proviola and the movie together **proxy** [32, Chapter 5] the behaviour of a proof assistant: the responses shown to the proof reader are stored (or cached) in the movie, after having been computed by the proof assistant.

A prototype implementation of the Proviola We implemented a prototype of the Proviola as an XSLT transformation from the movie into an HTML file containing embedded JavaScript. This page initially shows the commands within the movie, much like a proof script. Figure 3.6 illustrates this: when the reader places his cursor over a command, the corresponding response is revealed dynamically. The command pointed to is highlighted as a visual reminder. The prototype Proviola can be inspected at `http://mws.cs.ru.nl/proviola/examples.html`

3.4.2 Creating a Movie

Before a movie can be replayed, it must be created from a proof script by a tool we call a **camera**. Such *creation of a movie* is a new use case, shown in Figure 3.7.

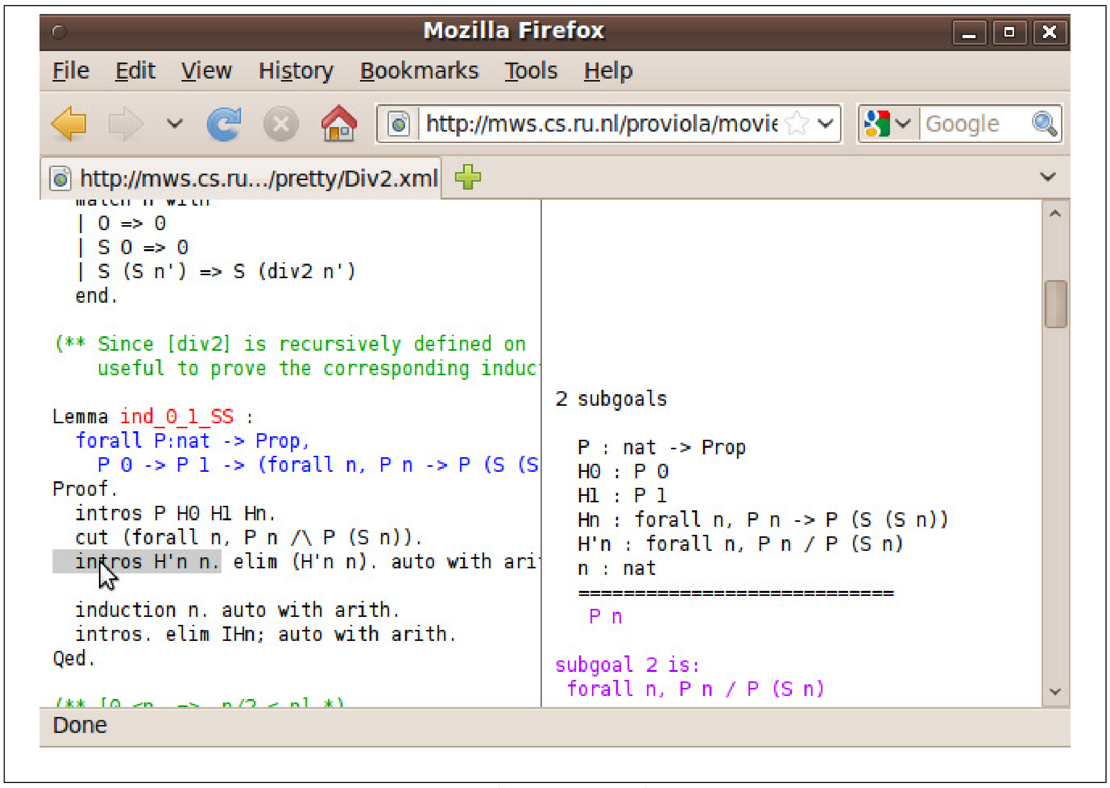
The displayed process is non-interactive: the user of the camera invokes it on a script, after which the tool does all the work, yielding a movie.

After it has been invoked, the camera parses the proof script given to it into separate commands. These commands are stored in a frame and sent to the proof assistant. When the proof assistant responds to a command, the response is recorded alongside the command. The frame is subsequently appended to the movie.

From the perspective of the proof assistant, the camera and the script behave like an actual proof author. In other words: in creating the movie, the camera and script together *proxy* the behaviour of a proof author.

Because the author holds the original script, it seems natural that she invokes the camera on it to obtain a movie, for distribution to readers. However, a reader might also play the role of cameraman, given access to the script.

Prototype implementation We implemented the camera as a client to the Coq, Isabelle and HOL light system. Each implementation utilises the interface that the proof assistant


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://mws.cs.ru.nl/proviola/movie
Google
http://mws.cs.ru.../pretty/Div2.xml
Done

```
  | O => 0
  | S O => 0
  | S (S n') => S (div2 n')
  end.

(** Since [div2] is recursively defined on
    useful to prove the corresponding induc

Lemma ind_0_1_SS :
  forall P:nat -> Prop,
    P 0 -> P 1 -> (forall n, P n -> P (S (S
Proof.
  intros P H0 H1 Hn.
  cut (forall n, P n /\ P (S n)).
  intros H'n n. elim (H'n n). auto with ari

  induction n. auto with arith.
  intros. elim IHn; auto with arith.
Qed.
```

```
2 subgoals

  P : nat -> Prop
  H0 : P 0
  H1 : P 1
  Hn : forall n, P n -> P (S (S n))
  H'n : forall n, P n / P (S n)
  n : nat
  ============================
   P n

subgoal 2 is:
 forall n, P n / P (S n)
```

Figure 3.6: Screenshot of a Proviola

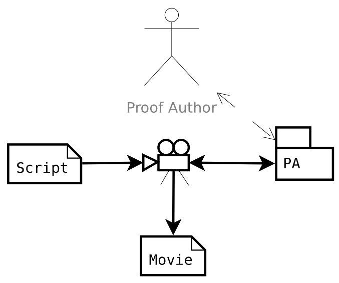


Figure 3.7: Creating a movie: camera and script proxy user behaviour

offers, generating a movie in the fixed format. In addition, we also implemented a camera as a client to the ProofWeb system, available at `http://mws.cs.ru.nl/proviola/camera/camera.html`. This implementation does not require a reader to install a proof assistant, while still making the movies himself (if the author neglected it), but requires the ProofWeb system to support the correct proof assistant version.

Agora creates the movie on-demand: when a document is requested, the system runs the parser step of the camera to create a rendered page for that document. When the reader points at a command on the page, the system then runs the proof assistant to fill in the responses and display the one pointed at. The responses are then cached for future retrieval. It is possible to generate the full movie when the document is stored in Agora, but this would cause longer delays before a page can be shown, with pages that are not inspected taking up extra time (and storage).

3.5 Rich Movies: Narratives in Coqdoc

The setup described thus far focused on plain text movies: they are obtained from a non-marked up proof script, and only add proof assistant responses to the display. To obtain a better exposition for a reader, the narrator can add a **narrative** to the movie: this describes a proof script in a possibly non-linear fashion with respect to that script.

There already exist approaches to narratives for proof scripts, falling broadly into two categories: either one can use specific syntax to write documentation inside the proof script (typically as comments). This documentation style is taken by tools inspired by JavaDoc [70]. It is intended to provide library documentation for software modules, by documenting for each function what arguments are expected and what the intended results (and side effects) are, as well as potential exceptions that can occur. As such, it does not provide a self-readable narrative, but mainly provides a documentation for each part of a library.

The second approach is known as *literate proving* [21] and requires the author to write both documentation and proof in tandem. This approach is inspired by Knuth's literate programming [56], which is a programming *paradigm* that encourages authors to write the explanation of a program, with the actual program code interleaved, non-linearly into the description. A pre-processor can then extract either the documentation or the program out of the full document.

Coqdoc is a Coq tool that is mostly used for the first approach. Distributed together with the Coq proof assistant, the tool produces a rendered (in HTML or in LaTeX) version of a proof script. This rendered document contains both a pretty-printed version of the commands, and special comments extracted from the proof script. These comments are taken as a narrative, and rendered as documentation. To provide some control over the appearance of the documentation, a light (Wikipedia-like) syntax is provided for marking up the narrative. While it supports a more literate style of proving, the documentation is still linear with respect to the proof.

As an example of the second approach, Aspinall, Lüth and Wolff [11] have developed an extension to their PG kit architecture based on literate proving. The extension is designed around a central document, that can be manipulated by the author and by tools. Example tools are a proof assistant, taking tactics and inserting proof state, or LaTeX-related tools, creating PDF out of the narrative. To insert proof assistant data inside the narrative, an author can use a command to insert a placeholder for the proof state, to be replaced later by actual proof assistant output.

Both of these approaches could produce HTML pages, but the pages are static renditions of the script, only containing pretty-printing to support communication and teaching: any prover output in the resulting document is there because it was inserted by the narrator, not because a reader requested it. In the next section, we make the Coqdoc-produced pages more dynamic, by adding a movie-reel.

Another interesting problem arises in both approaches when a new author wants to narrate a script that is provided 'read only': such a scenario, which might occur when documenting a third-party library, is not supported by either tool, although the PG kit approach might be adapted to support the scenario, by having the parts of a script represented by placeholders, just like the proof state is represented by a placeholder in the document.

3.5.1 Course notes

We have decided to focus on coursebooks for education using a proof assistant, and as a specific instance, we will look at the course notes by Pierce et al. for a course on Software Foundations taught at the University of Pennsylvania [72]. As the name implies, the course is not about proof assistants — although Coq is introduced during the course, but about the mathematical foundations of software and the semantics of programs.

The coursebook is entirely written as a set of Coq scripts, with the narrative as Coqdoc comments. Beyond the structuring in separate files, one for each chapter, the text is further structured in sections and subsections, by giving Coqdoc headers at the appropriate locations. This allows us to see the nesting of a single chapter as follows:

1. At the highest level we find a separation in sections. Each section can contain zero or more subsections.

2. At the deepest level of the document tree, the subsections have paragraphs as leaves. These leaves can be either commands to the proof assistant or paragraphs in the narrative.

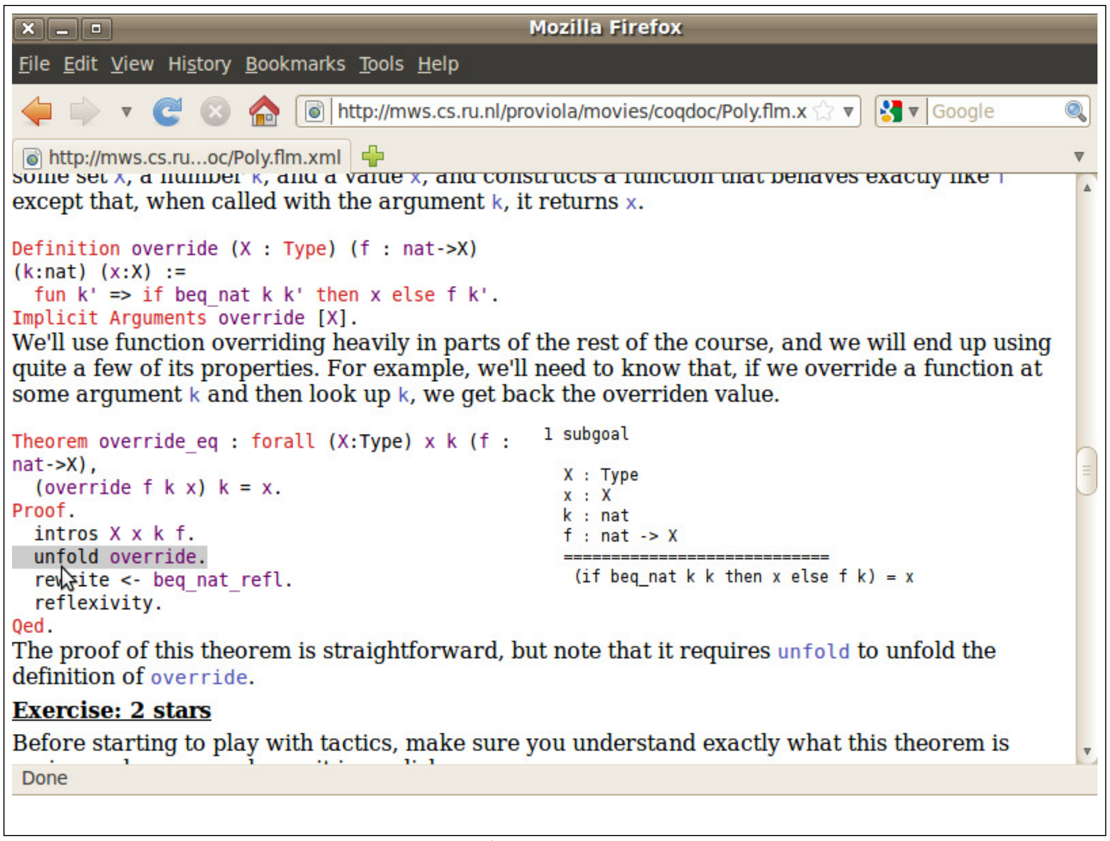
3. The proof script forms a special structure outside the structure of the text, that of a sequential set of commands interpretable by a proof assistant.

Chlipala has also written a coursebook, one on dependently typed programming [22], but we do not focus on it here, beyond the observation that he includes proof assistant output as part of the narrative, reinforcing our belief that it is desirable to perform the interleaving of movie and state rendering.

3.6 Enhancing movies with commentary

We now show how we can overlay our movies, representing the command structure of the proof script, on top of the Coqdoc-rendered document representing the narrative structure of the document. This is an enhancement of our original (plain text) movie data structure: by including rendered content the movie becomes more of a document than just a proof script. On the other hand, it is also an improvement over the text rendered by Coqdoc: instead of the static pages produced by this tool, the reader can request the results of a tactic to better understand why a tactic was chosen or what its use is.

The 'pretty' rendering of a movie, provided by Coqdoc, can easily be integrated in the movies. To do so, we created a tool to take commands from the frames and feed them to Coqdoc, which then outputs an HTML tree, containing more information about the intention of the command. In particular the tree can have nodes of the following types:


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://mws.cs.ru.nl/proviola/movies/coqdoc/Poly.flm.x
Google
http://mws.cs.ru...oc/Poly.flm.xml
Done

some set X, a number k, and a value x, and constructs a function that behaves exactly like f except that, when called with the argument k, it returns x.

```
Definition override (X : Type) (f : nat->X)
(k:nat) (x:X) :=
  fun k' => if beq_nat k k' then x else f k'.
Implicit Arguments override [X].
```

We'll use function overriding heavily in parts of the rest of the course, and we will end up using quite a few of its properties. For example, we'll need to know that, if we override a function at some argument k and then look up k, we get back the overriden value.

```
Theorem override_eq : forall (X:Type) x k (f :
nat->X),
  (override f k x) k = x.
Proof.
  intros X x k f.
  unfold override.
  rewrite <- beq_nat_refl.
  reflexivity.
Qed.
```

```
1 subgoal

  X : Type
  x : X
  k : nat
  f : nat -> X
  ============================
   (if beq_nat k k then x else f k) = x
```

The proof of this theorem is straightforward, but note that it requires unfold to unfold the definition of override.

Exercise: 2 stars

Before starting to play with tactics, make sure you understand exactly what this theorem is

Figure 3.8: A screenshot of the movie

- Documentation nodes, further structured in:
  - section headers, for different section levels,
  - narrative paragraphs, containing the text of the commentary.
- Code nodes. These nodes contain the tactics of the script.

The markup nodes produced by Coqdoc are stored in the frame as additional data, next to the command and respons. This data can be used for several purposes, especially for displaying the paragraphs nicely, but also for extracting semantic information, like we do in Chapter 4.

3.6.1 Rendering enhanced movies

Instead of displaying the plain text of a movie, we display the rendered text as created by Coqdoc instead. This is similar to the normal display of Coqdoc HTML pages, with the exception that placing a cursor on the code fragments dynamically displays the response to the command currently in focus.

Due to its dynamic nature, the best way to see the results is via the web, so we have provided a web page displaying these course notes dynamically at `http://mws.cs.ru.nl/proviola/examples.html` Despite the obvious limitations of including static screenshots here in order to illustrate a dynamic feature, Figure 3.8 displays the effect of placing the cursor on a tactic: the tactic is highlighted in grey, and the result of executing the tactic is displayed to the right of the code, inside the document. If the cursor moves to another tactic, the result of that tactic on the proof state would be displayed instead.

3.6.2 Scenes: Structuring a Narrative

The approach to Proviola-izing Coqdoc-rendered pages has one major drawback: Coqdoc works on a per-file basis it does not build create a rich rendering when provided with code snippets, which is what our Proviola provides in the form of frames. Not only is the hyperlinking lost, the system also does not provide the internal structure of a document. To remedy this situation, we turn the processing around: instead of first parsing the script and then rendering the fragments using Coqdoc, we first render the script, and parse the resulting HTML. This has some requirements:

- The rendering of code should be non-destructive, because the Proviola's camera cannot guess the missing commands. In practice, this means that Coqdoc should be instructed not to omit the proofs of lemmas and the script should not contain so-called 'hide' blocks: these blocks cause Coqdoc to drop all code that occurs inside a block.
- The camera should be capable of extracting code out of HTML nodes: in particular, it should remove markup from the code nodes without changing the code, and it should not consider documentation nodes as code.
- The movie data structure should represent the document. In particular, it should maintain the structuring into sections and the separation between code and documentation represented in the HTML document as `div` elements.

The last point is solved by enhancing the movie with a **scene** structure, already described on a high level in Chapter 2. Instead of just keeping the content of the document as a list of frames, we extend the movie with a tree structure in the form of scenes. Since scenes can freely refer to other scenes and frames, it is also possible to write narratives that are not linear with respect to the script they describe.

Narrative extracted from a script does have the major disadvantage compared to a normal text written for a coursebook: it is written inside a proof script and forms a linear interleaving with it: any explanatory text is sandwiched either side by tactics, and can therefore only easily refer to these tactics. When explaining a larger piece of code, this might not suffice, but instead require recalling a previous definition or lemma, or first showing the main theorem of a chapter, before going into the details of proving it.

Scenes form an alternative to this rigid structure. Like a *section* in a regular text, a scene is a unit of explanation, containing a text describing a part of the proof together with relevant references parts of the proof. That is, a scene has the following elements:

- structuring in the form of sub-scenes, and
- references to frames, which can either contain code or just marked up descriptions.

As mentioned in Chapter 2, scenes are equipped with a type, that guide the processing of the frames lying beneath a scene.

By this design, the scene structure forms a 'comb', including only specific frames of the movie (the 'teeth' of the comb) and narrating them in a continuous text (the 'handle'). This idea is illustrated in Figure 3.9.

We discern two tasks necessary for an author to compose a scene:

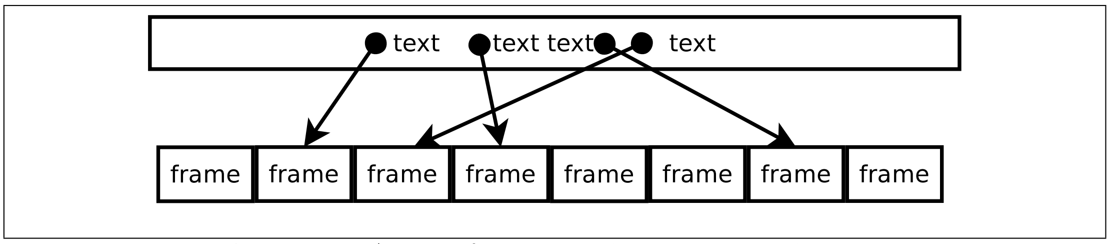


Figure 3.9: A scene forms a comb structure on the movie

1. select the frames that the scene describes (creating the teeth of the comb);
2. write the textual scenes, including references to the selected sub-scenes or frames (connecting the teeth with the handle).

When the author is writing the text, she should have a view on the frames she selected *only* and can then insert references in the text to the entries in this distilled list. To add or remove entries from the list of frames, the author should at any time be able to switch back to the selection task.

Structuring a movie into scenes can be done automatically, based on the Coqdoc output. We already mentioned that Coqdoc sorts nodes into code and documentation nodes, and that documentation nodes can be either paragraphs or section headers. So, to create a scene from a Coqdoc-annotated script, we only need to mimic the document structure using scenes and subscenes. Each documentation node in the script creates a scene, with the frames referred by it the code beyond it.

Agora's parser for Coqdoc HTML pages follows the page's grouping when creating scenes, and does not add any extra grouping. We have done a small experiment that creates scenes not only for textual structuring, but also for proofs of lemmas and theorems: because Coqdoc loses this information during processing, this entailed enhancing Coqdoc so it would create a `div` element for a lemma, which marks both the statement and the proof. The camera could then be extended to generate the corresponding scene. We did not pursue this further, as the patch to Coqdoc was difficult to maintain or apply to multiple versions of the tool.

3.7 Adding Commentary to a Proof

We believe scenes to be particularly useful for writing commentary *after* the proof script has been written. For this, the script should first be turned into a movie, and then further edited: selecting the frames and describing them in the narrative, as well as stringing scenes together into a **commentary track**.

We have experimented with an interface for writing the commentary track, but based on the scene structure and an initial prototype, we observe that the interface should provide for the following activities:

- writing the text of a scene;
- selecting the frames that appear in the scene;
- adding references to the selected frames; and
- string the scenes into a narrative.

Writing the actual text can be done in either a WYSIWYG editor or with some light markup language (as used in Wikipedia and Coqdoc), and should not introduce new HCI problems.

The first design decision we make is how to allow an author to group text into scenes. As the resulting document structure is a tree, a tree editor could be used for adding or removing or reordering scenes to the document, and selecting scenes for further editing. The main advantage of this approach is that the structure can be seen at a glance, and edited easily.

On the other hand, inferring the movie's structure when the author inserts a header might provide a faster editing workflow, as adding a new scene does not require her to switch to a different menu or editor.

These two approaches could be combined, inferring the document structure from commands typed in the editor and explicitly allowing an author to insert scenes or move scenes in a structure editor, actions which get translated to modifications of the text in the editor.

Selecting the frames to appear in a scene can be done by simply toggling them: the author is presented with a movie, in which she can click on the frames she wants to appear in the scene.

We have experimented with an interface that has a tree editor for adding scenes to a movie (only one level deep) and a rich text editor for writing the narrative per scene. To link this text with the code of the command, a third pane gives the author a view on the movie's commands and the responses, and allowing her to toggle frame inclusion by a click on the desired frames. A screenshot is shown in Figure 3.10; it can be experimented with at `http://mws.cs.ru.nl:8080/proofcomment`.

The implementation of this interface still forces the user in a rather restricted workflow: she would first need to add a scene, then alternate between typing and choosing code to be included.

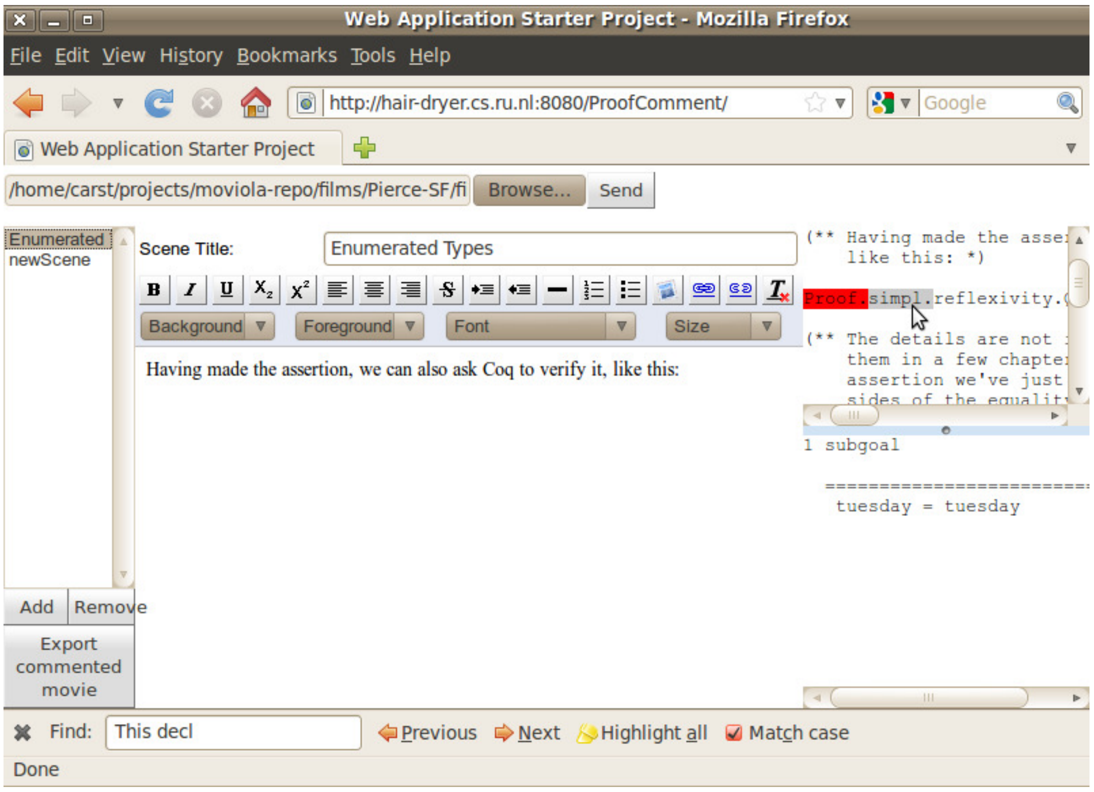
This interface imposes a rather strict workflow. A lighter-weight approach based on adding syntax to Agora's markup language is investigated in Chapter 4.

3.8 Towards Interactive Movies

We have added dynamic content to Coqdoc documents, but this does not make a proof document really *interactive*: a reader cannot change the underlying proof script when watching a movie, to try out, for example, an alternative approach to a given proof. In the context of course notes, a more important example is making exercises: when teaching with a proof assistant, this includes letting a student give missing definitions or complete certain proofs.

Providing the original proof script with the necessary definitions to a student is one way of giving exercises, but then he would need to switch to a proof assistant to actually do the exercise, instead of staying in the environment of the browser. Instead, we propose to add interactive elements on top of the movie's narrative. These interactive elements are scenes that can be edited using an editor such as the one described in Chapter 5.

For course notes, only specific parts of the movie are allowed to be edited by a reader, these sections need to be indicated by the author of the movie. However, when a movie is completely editable, it will no longer be necessary to generate a movie from a script; instead, an author can write a proof document in her browser, with the underlying machinery computing proof states and rendering the movie on-the-fly.



http://hair-dryer.cs.ru.nl:8080/ProofComment/

/home/carst/projects/moviola-repo/films/Pierce-SF/fi

Scene Title: Enumerated Types

Having made the assertion, we can also ask Coq to verify it, like this:

```
(** Having made the asse
    like this: *)

Proof. simpl. reflexivity.

(** The details are not
    them in a few chapte
    assertion we've just
    sides of the equalit
```

```
1 subgoal

  ========================
   tuesday = tuesday
```

Find: This decl

Figure 3.10: A screenshot of the commentary tool

3.8.1 Writing Editable scenes

An editable scene is a scene that contains some code frames to be edited by the reader after the movie is published. Adding such a feature requires:

- an interface option for the author through which she can mark which scenes can be edited later, and which should remain locked, and
- a proof assistant processing the commands the reader types in an exercise scene.

The author of a proof movie determines which scenes are editable and which scenes are locked: this can be done while she prepares a movie, by setting a property of the scene, comparable to making a file read-only in the file system. How the property is set depends on the editor style chosen: a WYSIWYG editor might provide it as an option in a context menu, while a markup language could allow some meta-command for setting the attribute of a scene. In particular, the sources for the Software Foundations course notes have explicitly marked "exercise" blocks, which are used for their document preparation, but which could also be harvested by our camera: this would make the addition of an exercise scene semi-automatic.

3.8.2 Interacting with Editable Scenes

Once we have integrated the notion of an editable scene within the movie's data structure, the display of the movie needs to accommodate editing these scenes. The first step in this would be indicating to the student that a scene is editable, for example by providing an edit button next to the scene, and by including a proof assistant-backed editor for filling out the exercise.

Chapter 5 describes such an editor, which fulfills the first four steps of the following use case.

1. The student clicks the 'edit button'.
2. The movie's server brings a proof assistant into the state necessary for doing the exercise.
3. The editor is shown to the student, including the proof assistant's state (context and goals) for the exercise.
4. In the editor, the student types commands, which update the proof assistant's state.
5. If the student solves the exercise, it is stored, if he abandons it, the exercise gets abandoned.

The main open problem is handling the proof assistant state efficiently: before the editor is shown, quite some computation is necessary to bring the proof assistant into the right state. How to handle this computation remains an open question, but we have some ideas on how to tackle it:

- At the moment the document is shown to the student, also feed it to the proof assistant as a background process, stopping at the first exercise. This is a naive, but easily implemented solution, that does not account for exercises being skipped or abandoned.
- To handle a student skipping an exercise, we could tacitly insert a command like `Admitted`, which assumes the open goal as an axiom, for every exercise. Once a student has solved it, we then remove the admission. This would work for Coq, but not all proof assistant s support an `Admitted`-like construct, so this solution cannot be easily adapted to other

proof assistant s that we might want to adapt our technology to. Apart from that, the computation to get to the focused exercise might betoo slow, as the student might start with the last exercise, requiring the entire chapter to be sent to the proof assistant in order to start the exercise.

- We could be smarter about the inter-proof dependencies: most proof assistant s interpret the script as a linear sequence, each command depending on all of the previous. This is not always the case, however, especially for exercises, where the proof structure resembles a tree, with the exercise being leaves depending on the content of the explanation above it. We could exploit this structure by only checking the path to the leaf that is focused, instead of all subtrees. To actually make this work, either the proof assistant needs to be more permissive about the proof structure, or external tools could submit only the commands that are necessary to get to the selected leaf to a proof assistant.
- Finally, we observe that a large part of the proof does not change when a student starts an exercise: the proof script that is part of the explanation is locked by the author, and would not need to be rechecked each time an exercise is attempted. So, we could 'restore' a proof session starting at the exercise. This approach works for systems implemented in Poly/ML (such as Isabelle), but we have not investigated the space requirements for keeping a reasonably-sized set of exercises in a 'frozen' state.

Once we have the machinery for checking proof text in place, we can further investigate how to support (unsupervised) e-learning using proof assistant s: by checking the 'correctness' of a solution (*e.g.* that it does not contain commands that automatically find proofs) and by giving hints to students on how to solve a given exercise.

3.9 Related work

Leading up to this chapter, we created a dynamic version of the Software Foundations course notes [72]. We have applied our techniques to create handouts for a proof assistant and type theory course Geuvers teaches at the Eindhoven University of Technology. Other documents that we could transform are the Coq tutorial by Huet et al. [46] and the tutorial by Bertot [17].

Several approaches exist based around a central document for formal proof, similar to our movie, of which we have already mentioned the PG kit approach by Aspinall, Lüth and Wolff [11], and the tmEgg experimental document-oriented Coq plugin for TEXmacs by Mamane and Geuvers [35]. Hinze and Löh's lhs2TEX [64] which is meant for literate Haskell programming has been adapted to allow writing literal proof documents, from which both Coq code and LATEX documentation can be extracted: this approach is an implementation of Knuth's literal programming paradigm. The coq-tex tool in the Coq distribution does something similar, executing Coq commands within a LATEX document and returning the output in a LATEXsource file for further processing by the author: the output has to be placed by the author, and not by the reader.

These approaches are mainly used for writing proof and documentation together, while our movie allows an author to first write a proof script, and then create a dynamic presentation of this script. The presentation can then be used in a narration of the proof.

Nordström has suggested [69] using dependent type theory to enforce syntactic wellformedness of books and articles, 'live' documents, programs, and formal proofs in a unified way. In particular, his notion of typed placeholders could be used to represent exercises in a online coursebook: such placeholders represent the form of a solution as a dependent type, allowing the system to verify whether an exercise is syntactically correct. On the other hand, it allows the system to verify a document that contains placeholders, meaning it is possible to write a movie that contains open exercises, but that can be still be verified.

3.10 Conclusions

We have shown how we can make on-line coursebooks using a proof assistant more dynamic: by adding the proof assistant's output to the document and showing it when requested by the student reading the book. Constructing these dynamic books is the result of combining two techniques: our previous work on creating movies out of a proof script, and the addition of markup and commentary to a proof document using tools such as Coqdoc.

We have further sketched how a commentary track can be added to proof documents, and how to add interactive elements to these documents.

The techniques for creating the dynamic, non-interactive documents have been applied to the course notes for a "Software Foundations" course and have been received with great enthusiasm by the authors of these notes. This shows that the documents we create with the described tooling add value to the Coqdoc output, and gives motivation for improving the workflow and output.

The data structure and the rendering engine have been used in Agora to represent the structure of proof scripts. This allows several workflows to be defined that not only use the dynamic display of proofs, but the structure as well. These workflows are described in the rest of this thesis.

Chapter 4

Point and Write

This chapter is based on the paper “Point-and-Write — Documenting Formal Mathematics by Reference” by Tankink, Lange and Urban [86]. The modifications are minor textual changes, and some extended explanations.

4.1 Introduction

Formal, computer-verified, mathematics has been informally discussed and written about for some fifty years: on dedicated mailing lists [27, 67, 47], in conference and journal articles, online manuals, tutorials and courses, and in community Wikis [23, 68] and blogs [89].

In such informal writings, it is common to include and mix formal definitions, theorems, proofs and their outlines, and sometimes whole sections of formal articles. Such formal “islands” in a text do not have to follow any particular logical order, and can mix content from different articles, libraries, and even content based on different proof assistants. In this respect, the collection of such formal fragments in a particular text is often *informal*, because the fragments do not have to share and form a unifiable, linear, and complete *formal context*.

In a Web setting, however, such pieces of formal code can be equipped with semantic and presentation functions that make formal mathematics attractive and unique. Such functions range from “passive” markup, like (hyper)linking symbols to their precise definitions in HTML-ized formal libraries, detailed and layered explanations of implicit parts of reasoning (goals, types, subproofs, etc.), to more “active”, like direct editing, re-verification, and HTML-ization of the underlying formal fragment in its proper context, and using the formal code for querying semantic search engines and automated reasoning tools, such as the one described by Sutcliffe and Urban [93].

In this chapter, we describe a use case of an author writing such an informal text: she gives references (**points**) to (fragments of) formalization on the Web, and then describes (**writes** about) them in a natural language narrative, documenting the formal islands. This use case is described in Section 4.2.

We further give the design (and implementation) details of tools that support this use case in a light-weight manner, based on HTML presentations of formal mathematics. The author can write pointers to formal objects in a special syntax (described in Section 4.3), which get resolved to the objects when rendering the narrative. To follow the pointers, our tools equip HTML pages for formalizations with annotations describing *what* a particular HTML fragment represents (Section 4.5): in the pointer-analogy, these annotations form the signposts. The annotations are drawn from suitable RDF vocabularies,[1] described in Section 4.4.

We show an implementation of the mechanisms in the Agora prototype,[2] described in Section 4.6. The actual *rendering* of the final page with its inclusions gives rise to several issues that we do not consider here. We discuss some of these rendering issues and how we handle them in our implementation, but these decisions were not made systematically: we focus here on the author's use case of writing the references, and provide the prototype as a proof of existence. As we show in Chapter 6, this prototype is powerful enough to handle a large formalization, although this required some ad hoc modifications to the prototype. The main reason that we still consider the tool a prototype lies in usability issues as well as some engineering issues.

To our knowledge, there is no system that provides similar functionality for inclusion: there are alternatives for including formal text, as well as alternative syntaxes, which we will compare with our approach in the relevant sections. Additionally, the MoWGLI project has developed some techniques for rendering formal proofs with informal narratives, which are compared to in Section 4.6.

4.2 Describing *and* Including Formal Text

The techniques described in this chapter are mainly driven by a single use case, that of an author writing a description (a "narrative") of a development in formal, computer-verified, mathematics. In this work, we assume that the author writes this narrative for publication in a Wiki, although the use case could also be applied for more traditional authoring, in a language like LaTeX.

4.2.1 Use Case

While writing a natural language narrative, the author will eventually want to include snippets of formal code: for example to illustrate a particular implementation technique or to compare a formalization approach with a different one, possibly in a different formal language. An advanced example is Section 5.3 of [14] rendered in Agora.[3]

Because we allow for including formal text from the Web, there is a wrinkle we have to iron out when supporting this use case, but before we get to that, we describe typical steps an author can carry out while executing the use case:

Formalization An author works on a formalization effort in some system and puts (parts of) this formalization on the Web, preferably on the Wiki. We will refer to the results of these efforts as **source texts**.

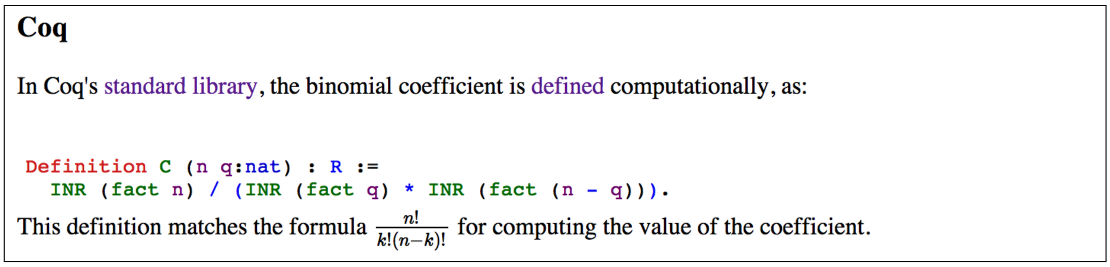
Natural language description Sometime before, during or after the formalization, the author gives a natural language account of the effort on the Wiki. As mentioned, we assume she writes this in a markup language suitable for Wikis, extended with facilities for writing mathematical formulae.[4] We refer to the resulting description as a **narrative**.

[1]The RDF data model (Resource Description Framework) essentially allows for identifying any thing ("resource") of interest by a URI, giving it a type, attaching data to it, and representing its relations to other resources [1].

[2]`http://mws.cs.ru.nl/agora/`

[3]`http://mws.cs.ru.nl/agora/cicm_sandbox/CCL/`

[4]The specifics of suitable languages for writing in Wikis for formal mathematics are not a subject of this chapter; we refer to [62] for an overview.



Coq

In Coq's standard library, the binomial coefficient is defined computationally, as:

```
Definition C (n q:nat) : R :=
  INR (fact n) / (INR (fact q) * INR (fact (n - q))).
```

This definition matches the formula $\frac{n!}{k!(n-k)!}$ for computing the value of the coefficient.

Figure 4.1: Example of informal narrative with formal snippets

Including formalizations into the narrative In the natural language description, the author includes some of the formal artifacts of her effort, and possibly some of the formalizations by other authors. These inclusions need to look attractive (by being marked up) and should not be changed from the source: the source represents a verifiable piece of mathematics, and a reader should be able to ascertain himself that nothing was lost in transition.

These steps are not necessarily carried out in order, and can be carried out by different authors or iteratively. In particular, the formal text included in the narrative does not have to originate from the author or her collaborators, but could be from a development that serves as competition or inspiration.

The end results of the workflow are pages like the one shown (in part) in Figure 4.1: it includes a narrative written in natural language (including hyperlinks and markup of formulae) and displays formal definitions marked up as code.

Because the source texts are stored on the Web, we consider their content to be *fluid*: subject to change at any particular moment, but not under control of the author. This implies that the mechanism for including formal text should be robust against as much change as possible.

To determine how we can support this workflow, we first survey the existing methods that are suitable for including formal mathematics.

4.2.2 Alternatives for Inclusion

Typical options for including formal mathematics—or any other type of code—, when working with a document authoring tool like LaTeX include:

1. **Referral:** place the code on some Web page and refer readers to that page from the document (by giving the URL),

2. **Inclusion:** include and format the source code files as listings, e.g. using the LaTeX package listings [45],

3. **Literate proving:** the more extreme variant of (2): write the article in a literate style, and extract both formal code and marked up text from it,

4. **Copy-paste:** manually copy-paste the code into the document.

All of these options have their own problems for our use case.

1. Referral collides with the desire for *juxtaposability* [18]: a reader should not have to switch between pages to look at the referred code and the text that refers to it. Instead, he should be able to read the island within the context of the narrative.

2. We certainly want the author to be able to include code, but most of the tools only allow her to refer to the code by *location*, instead of a more semantic means: she can give a range of lines (or character offsets) in a file, but cannot write "include the Fundamental Theorem of Algebra, and its proof".

3. Literate proving [21] is a way to tackle code inclusion, but it does not solve the use case: it requires the author to shift her methods from writing code and article separately to writing both aspects interleaved. While the approach given by Cairns and Gow does allow an author to mark up a formal proof with literate elements after the proof has been written, this still takes the proof as the leading subject instead of the article. In particular, this means that it is not possible to change the order in which a formalization is displayed.

   It also does not allow an author to include existing external code for citation (without copy-pasting) and does not allow her to write a document including only snippets of formal code. These cases can arise where a lot of setup and auxiliary lemmas are necessary for formalizing a theorem, but only the theorem itself is the main focus of a paper. Typical literate programming setups provide mechanisms for hiding code fragments, but we prefer to take an inclusive instead of an exclusive view on the authoring process: the former seems to be more in line with actual practices in the interactive theorem proving community (see, for example, the proceedings of the Interactive Theorem Proving conference [94]).

4. Copy-pasting code has the traditional problem of maintaining consistency: if the source file is changed, the citation should change as well. On the positive side, it does not require much effort to implement, apart from adding facilities for marking up code, which can be reused for in-line (new) code. To make the implementation threshold even lower, the listings LaTeX package previously mentioned also supports marking up copy-pasted code.

The shortcomings of these methods mean we need to design a system providing the following facilities:

Requirement 1. *A* syntax *for writing, in a natural language document, references to parts of a formal text, possibly outside of the referring text, and a mechanism for including the referred objects verbatim in a rendered version of the natural language text.*

Requirement 2. *A method for annotating parts of formal texts, so they can be referenced by narratives.*

The rest of this chapter gives our approach to satisfying these two requirements, demonstrating how they interact, and gives a tour of our working implementation.[5]

4.3 Syntax for Referring

We first focus on *how* the author can write references to formal content. Below, we discuss considerations that guided the design of this syntax. The considerations are partially based on the goals for a common Wiki syntax [76].

[5] `http://mws.cs.ru.nl/agora/cicm_sandbox`

4.3.1 Requirements on syntax

Simple To encourage its use, the syntax should not be too elaborate. An example of a short enough syntax is the hyperlink syntax in most Wiki systems: only four characters surrounding the link, `[[` and `]]`.

Collision free The syntax should not easily be 'mistyped': it should not be part of the syntax already used for markup, and not likely used in a natural language narrative.

Readable It should be recognizable in the source of the narrative, to support authors in learning a new syntax and making the source readable.

Familiar We do not intend to reinvent the wheel, but want to adapt existing syntax to suit our needs. This also keeps the syntax readable: when the base syntax is already known to an author, it should be clear to her how this syntax works in the context of referring to formal text. Because keeping things similar but not completely equal could cause confusion, it also requires us not to deviate the behavior too much from the original syntax.

In resolving these requirements on the syntax, we need to consider the context in which it will be used. In our proposed use case, the syntax will be used within a Wiki, so we prefer a syntax that fits with the markup families used for Wiki systems. It should be possible to extend a different markup language (for example, LaTeX or literate comments for a formal system) with the reference syntax, but this requires reconsidering the decisions we make here, to better fit those contexts.

Considering that we want to base the reference syntax on existing mechanisms (in line with the familiarity requirement), there are three basic options to use as a basis: import statements like those used in LaTeX (or, programming and formal languages), Wiki-style hyperlinks, and Isabelle/Isar's antiquotation syntax.

Each of these is considered in the rest of this section, and tested against the requirements stated before.

LaTeX-style include statements. The purpose of these statements in a LaTeX document is to include the content of a file in another, before rendering the containing file. While there are several such commands, the most recognizable is `\input{file.tex}`, which includes the contents of `file.tex` verbatim in the place of the input statement, before rendering the page. The command does not allow inclusion of file fragments, but could be modified to allow this, by adding an options that allows specifying the part of the content to be included. As a concrete example of this approach, the listings package mentioned earlier has the option to include file fragments by giving a line offset, but not a pointer to an object. In theory, the language can be extended with a semantic map of objects to file lines, but this requires an external processor for each language that needs to be included, and the post-processing needs to be done when a (static) PDF is rendered for the source document. Because the input statement is tied to including a file (or a web page), we cannot extend this easily to indirect references (which use queries, or an external resolution service).

Syntactically, the input statements are recognizable by LaTeX users or users of a formal language that uses inclusions, but the statements are rather long: if the author wants to use them more often, it might become tedious to write.

The MediaWiki engine has a similar syntax for including entire pages.[6] To include fragments of pages, however, one either needs to factor out these fragments of the source text and

[6] `http://www.mediawiki.org/wiki/Transclusion`

include them both in the source text and the referring text, or mark fragments of the source page which will be included. Both do not give the author of a narrative fine-grained control over inclusion.

An extension to these inclusions[7] allows inclusion of sections. This mechanism is a valid option for adaption, but if we would want to support the informal inclusion that this mechanism is intended to support, it would be difficult for a reader of the source code to distinguish between an informal inclusion and a formal one.

Wiki-style hyperlinks. These cross-reference statements are not hard to learn and short, but also using them for inclusion can overload the author's understanding of the markup commands: if she already knows how to use hyperlinks, she needs to learn how to write and recognize links that include formal objects. It is hard to distinguish between 'normal' hyperlinks and links that cause an included formal entity to render on the page.

Antiquotations. Isabelle/Isar [100] uses antiquotations to allow the author of Isar proof documents to write natural-language, marked-up snippets in a formal document (the 'quotation' from formal to informal), while including formal content in these snippets (the 'antiquotation' of informal back to formal): these antiquotations are written `@{type [options] syntax }`, where `type` declares what kind of syntax the formal system can expect, the `syntax` specifying the formal content, and the `options` defining how the results should be rendered. The formal system interprets these snippets and reinserts the results into the marked-up text.

For example, the antiquotation `@{term [show_types] "% x y. x"}` would ask Isabelle to type-check the term $\lambda x\ y.\ x$ (% is Isabelle's ASCII shorthand for λ, which can also be written directly using a Unicode-aware editor) in the context where it appears and reinserts the term annotated with its type: it inserts the output `λ(x::'a) y::'b. x`.

Another example is `@{thm foo}` which inserts the statement of the theorem labeled `foo` in the marked up text. The syntax also provides an option to insert the label `foo`, but only if the labelled entity is correct. This allows the author to mark up a text that has references to verified theorems, both by giving the name and by statement.

4.3.2 Resulting Syntax

From the options listed above, the antiquotation mechanism is closest to what we want: it allows the inclusion of formal text within an informal environment, relying on an external (formal) system to provide the final rendering. There are some differences in the approaches that require some further consideration.

Context In Isar, the informal fragments are part of a formal document, which gives the context in which to evaluate the formal content. In our use case, there is no formal context: the informal and the formal documents are strictly separated, so the formal text has to exist already, and is only referred to from a natural-language document.

We could provide an extension that allows the author to specify the formal context in which formal text is evaluated. This would allow her to write new examples based on an existing formalization, or combine literate and non-literate approaches. This is an appealing idea, but beyond the scope of this thesis.

Feedback In our use case, the natural language text only refers to the formal text, and does not feed back any formal content into the formal document. In Isar, it is possible to

[7] http://www.mediawiki.org/wiki/Extension:Labeled_Section_Transclusion

prove new lemmas in an antiquotation, but Wenzel notes in his thesis [100, page 65] that antiquotations printing well-typed terms, propositions and theorems are the most important ones in practice, at least for documentation purposes: later versions of Isabelle also provide antiquotations from ML fragments embedded within the proof document, but we do do not consider this here.

With these considerations in mind, we adopt the following syntax, based on the antiquotations: `@{ type reference [options] }`. The main element is `reference`, which is either a path in the Wiki or an external URL, pointing to a formal entity of the given `type`. We will discuss possible types in the next section.

The `options` element instructs the renderer of the Wiki about how to render the included entity. Compared to Isar, it has swapped positions with `reference` because it provides rendering settings, and no instructions to a formal tool. This means that they are processed last, after the reference has been processed to an object. Possible uses include flagging whether or not to include the proof of a theorem, or the level of detail that should be shown when including a snippet.

The `reference` points to an annotated object, by giving the location of the document it occurs in and the name given in the annotation for that object. The `type` corresponds to the type in the annotation of the object: it serves as a disambiguation mechanism, but can be enforced in a more strict manner. If the system cannot find a reference of the given type, it should fail in a user friendly way: in our implementation, we inline `reference` in the output, marked up to show it is not found. Inspired by MediaWiki, we color it red, and put a question mark after it. An addition to this would be to make this rendering a link, through which the author can write the formal reference, or search for similar objects.

The antiquotation for the Coq code in Figure 4.1, is `@{oo:Definition CoqBinomialCoefficient#C}`. It points to the `Definition C`, found in the location (a Wiki page) `CoqBinomialCoefficient`. This reference gets resolved into the HTML shown in the screenshot in Figure 4.1.

4.4 Annotation of Types and Content

For transforming antiquotations to HTML, we could implement ad hoc reference resolution mechanisms specific to particular formal systems. Then, any new formal system would require building another specific dereferencing implementation from scratch. We present a more scalable approach with lower requirements for formal systems. We enrich the HTML export of the formal texts with annotations, which clearly mark the elements that authors can refer to. The Wiki can resolve them in a uniform way: when an author writes an antiquotation, the system can dereference it to the annotated HTML, without further requirements on the structure of the underlying formal texts.

This section introduces the two main kinds of annotations that are relevant here; the next section explains how to put them into formal texts. Firstly, we are interested in annotating an item of formalized mathematics with its mathematical *type* (such as definition, theorem, proof), which allows an author to refer to these annotations in her references, and allows the rendering process to collect the correct island by searching for a labelled item of the given type. Secondly, we want to annotate items by pointing to related *content* (such as pointing from a formalized proof to the Wikipedia article that gives an informal account of the same proof), which allows an author to refer not only to the strict name of an entity, but also to a more generic concept. Type annotation requires a suitable annotation *vocabulary*, whereas we had to identify suitable *datasets* that can be used for labeling items with contents.

4.4.1 The Type Vocabulary of the OMDoc Ontology

The OMDoc ontology provides a wide supply of types of mathematical knowledge items, as well as types of *relations* between them, e.g. that a proof proves a theorem [60, 61]. It is a reimplementation of the conceptual model of the OMDoc XML markup language [59] for the purpose of providing semantic Web applications with a vocabulary of structures of mathematical knowledge. We use the terms "ontology" and "vocabulary" synonymously. When annotating documents or data, one usually speaks of an "annotation *vocabulary*", whereas using the term "ontology" emphasizes that a vocabulary has been implemented in an ontology language (here: OWL) having a formal, logic-based semantics. It is thus one possible vocabulary (see [61] for others) applicable to the lightweight annotation of mathematical resources on the Web desired here, without the need to translate them from their original representation to OMDoc XML.

The OMDoc language has originally been designed for exchanging formalizations across systems, e.g. for structured specification, automated verification, and interactive theorem proving [59]. OMDoc covers a large subset of the concepts of common languages for formalized mathematics, such as Mizar or Coq; in fact, partial translations of these specific languages to OMDoc have been implemented (see, e.g., [13]).

The OMDoc *ontology* covers most of the concepts that the OMDoc language provides for mathematical statements, structured proofs, and theories. Item types include *Theory*, *Symbol* [Declaration], *Definition*, *Assertion* (having subtypes such as *Theorem* or *Lemma*), and *Proof*; types of relations between such items include *Theory–homeTheoryOf–<any type of statement>*, *Symbol–hasDefinition–Definition*, and *Proof–proves–Theorem*. The ontology leaves the representation of document structures without a mathematical semantics, such as sections within a theory that have not explicitly been formalized as subtheories, to dedicated document ontologies (cf. [61]).

4.4.2 Datasets for Content Annotation

Our main use case for content annotation (annotating a formal entity with a pointer to related information, in particular what content it contains) is annotating formalizations with related informal representations, but added-value services may still benefit from such informal representations having a *partial* formal semantics. Consider linking a formalized proof to a Wikipedia article that explains a sketch and the historical or application context of the proof.[8] The information in the Wikipedia article (such as the year in which the proof was published) is not immediately comprehensible to Web services or search engines. For this purpose, DBpedia[9] makes the contents of Wikipedia available as a linked open dataset.[10]

Further suitable targets for content annotation of mathematical formalizations – albeit not yet available as machine-comprehensible linked open data –, include the PlanetMath encyclopedia, the similar ProofWiki, and Wolfram's MathWorld.[11]

In the interest of machine-comprehensibility, the links from the annotated sources to the target dataset should be *typed*. The two most widely used link types, which are also widely supported by linked data clients, are *rdfs:seeAlso* (a generic catch-all, which linked data clients usually follow to gather more information) and *owl:sameAs* (asserting that all properties asserted about the source also hold for the target, and vice versa). The OMDoc ontology

[8]The Wikipedia category "Article proofs" lists such articles; see `http://en.wikipedia.org/wiki/Category:Article_proofs`.
[9]`http://dbpedia.org`
[10]A collection of RDF descriptions accessible by dereferencing their identifiers [44]
[11]See `http://www.planetmath.org`, `http://www.proofwiki.org`, and `http://mathworld.wolfram.com`, respectively.

furthermore provides the link type *formalizes* for linking from a formalized knowledge item to an informal item that verbalizes the former, and the inverse type *verbalizes*.

4.5 Annotating Formal Texts

Now that we have established *what* to annotate formal texts with, we need to look at the *how*. Considering that the formal documents are stored on the Web, we assume that each document has an HTML representation. Indeed, the systems we support in our prototype each have some way of generating appropriate type annotations.

Text parts are annotated by enclosing them into HTML elements that carry the annotations as RDFa annotations. RDFa is a set of attributes that allows for embedding RDF graphs into XML or HTML [2]. For identifying the annotated resources by URIs, as required by RDF, we reuse the identifiers of the original formalization.

Desired Results. Regardless of the exact details of the formal systems involved, and their output, the annotation process generally yields HTML+RDFa, which uses the OMDoc ontology (cf. Section 4.4.1) as a vocabulary. For example, if the formal document contains an HTML rendition of the Binomial Theorem, we expect the following result (where the prefix *oo:* has been bound to the URI of the OMDoc ontology[12]):

```
<span typeof="oo:Theorem" about="#BinomialTheorem">...</span>
<span typeof="oo:Proof"><span rel="oo:proves" resource="#BinomialTheorem"/>
...</span>
```

The "..." in this listing represent the original HTML rendition of the formal text, possibly including the information that was used to infer the annotations now captured by the RDFa attributes. `@about` assigns a URI to the annotated resource; here, we use fragment identifiers within the HTML document.

In this example, we wrap the existing HTML in `span` elements, because in most cases, this preserves the original rendering of the source text. In particular, empty `span`s, as typically used when there is no other HTML element around that could reasonably carry some RDFa annotation, are invisible in the browser. If the HTML of the source text contains `div` elements, it becomes necessary to wrap the fragment in a `div` instead of a `span`. In Agora's data model, we can add annotations to scenes, which fits the model of scenes as semantical entities.

Mizar texts. Mizar processing consists of several passes, similar in spirit to those used in compilation of languages like Pascal and C. The communication between the main three passes (parsing, semantic analysis, and proof checking) is likewise file-based. Since 2004, Mizar has been using XML as its native format for storing the result of the semantic analysis [90]. This XML form has been since used for producing disambiguated (linked) HTML presentation of Mizar texts, translating Mizar texts to ATP formats, and as an input for a number of other systems. The use of XML as a native format guarantees that it remains up-to-date and usable for such external uses: the external tools use the XML representation as a basis, and do not need to assume the internals of Mizar, which might change without warning.

This encoding has been gradually enriched to contain important presentational information (e.g., the original names of variables, the original syntax of formulas before normalization, etc.), and also to contain additional information that is useful for understanding of the Mizar texts, and ATP and Wiki functions [93, 91] over them (e.g., showing the thesis computed by

[12]`http://omdoc.org/ontology#`

the system after each natural deduction step, linking to ATP calls/explanations, and section editing in a Wiki).

We implemented the RDF annotation of Mizar articles as a part of the XSL transformation that creates HTML from the Mizar semantic XML format. While the OMDoc ontology defines vocabulary that seems suitable also for many Mizar internal proof steps, the current Mizar implementation only annotates the main top-level Mizar items, together with the top-level proofs. Even with this limitation this has already resulted in about 160000 annotations exported from the whole MML,[13] which is more than enough for testing the Agora system. The existing Mizar HTML namespace was re-used for the names of the exported items, such that, for example, the Brouwer Fixed Point Theorem:[14]

```
:: $N Brouwer Fixed Point Theorem
theorem Th14:
  for r being non negative (real number), o being Point of TOP-REAL 2,
      f being continuous Function of Tdisk(o,r), Tdisk(o,r)
  holds f has_a_fixpoint
proof ...
```

gets annotated as[15]

```
<div about="#T14" typeof="oo:Theorem">
  <span rel="owl:sameAs"
       resource="http://dbpedia.org/resource/Brouwer_Fixed_Point_Theorem"/> ...
  <div about="#PF23" typeof="oo:Proof"><span rel="oo:proves" resource="#T14"/> ... </div>
</div>
```

Apart from the appropriate annotations of the theorem and its proof, an additional link labeled *owl:sameAs* is produced to the DBpedia (Wikipedia) “Brouwer_Fixed_Point_Theorem” resource. Such links are produced for all Mizar theorems and concepts for which the author defined a long (typically well-known) name using the Mizar `::$N` pragma. Such pragmas provide a way for the users to link the formalizations to Wikipedia (DBpedia, ProofWiki, PlanetMath, etc.), and the links allow the data consumers (like Agora) to automatically mix together different (Mizar, Coq, etc.) formalizations using DBpedia as the common namespace.

Coq texts. Coq has access to type information when verifying a document. This information is written into a *globalization* file, which lists types and cross-references on a line/character-offset basis. Coq’s HTML renderer, Coqdoc, processes this information to generate hyperlinks between pages, and style parts of the document according to the given types. Coqdoc is implemented as a single-pass scanner and lexer, which reads a Coq proof script and outputs HTML (or LaTeX) as part of the lexing process.

The resulting HTML page contains the information we defined in Section 4.4, but serves this in an unstructured way: individual elements of the text get wrapped in `span` elements corresponding to their syntactical class, and there is no further grouping of this sequence of `spans` in a more logical entity (e.g. a `<div id="poly_id" class="lemma">...</div>`), which would be addressable from our syntax. In particular, it puts the name anchor around a theorem’s label, instead of around the entire group.

For example, the following Coq code:

```
  Lemma poly_id: forall a, a → a.
```

gets translated into the following HTML fragment (truncated for brevity and whitespace added for legibility):

[13]MML version 4.178.1142 was used, see `http://mizar.cs.ualberta.ca/~mptp/7.12.02_4.178.1142/html/`

[14]`http://mizar.cs.ualberta.ca/~mptp/7.12.02_4.178.1142/html/brouwer.html#T14`

[15]`T14` is a *unique internal* Mizar identifier denoting the theorem. `Th14` is a (possibly non-unique) *user-level* identifier (e.g., `Brouwer` or `SK300` would result in `T14` too).

```
<span class="id" type="keyword">Lemma</span>
<a name="poly_id"><span class="id" type="lemma">poly_id</span>
</a>...
```

Aside from the fact that the HTML is not valid (`span` elements do not allow `@type` attributes), it has the main ingredients we are interested in extracting for annotation (type and name), but no indication that the keyword `Lemma`, the identifier `poly_id` following it, and the statement `forall a, a→a.` are related. The problem worsens for proofs: blocks of commands are not indicated as a proof, and there is no explicit relation between a statement and its proof, except for the fact that a proof always directly follows a statement. This makes the Coqdoc-generated HTML not directly suitable for our purpose; we need three steps of post-processing:

1. **Group objects:** The first step we take is grouping the 'forest' of markup elements that constitutes a command for Coq in a single element. This means parsing the text within the markup, and gathering the elements containing a full command in a new element.

2. **Export type information:** We then export the type information from Coq to the new element. We extract this information from the `@type`, derive the corresponding OMDoc type from it, and put that into an RDFa `@typeof` attribute.

3. **Explicit subject identification:** The final step is extracting `@name` and putting it into `@about`, thus reusing it as a subject URI.

After post-processing, we obtain the desired annotated tree, containing the HTML generated by Coqdoc.

The approach introduced here does not yet allow us to indicate the proof blocks, for which we do need to modify Coqdoc. The adaption is fairly straightforward: each time the tool notices a keyword starting a proof, it outputs the start of a new span, `<span typeof="oo:Proof">`. Similarly, the adapted tool outputs `</span>` when encountering a keyword signaling the end of a proof. Coqdoc contains functionality that recognizes the beginning and end of a proof, using it to hide a proof when an author wants to generate an HTML page containing only the statements of theorems; we can use these code fragments to generate our desired output.

Isar texts. For Isar texts, we have experimented with the annotation process. We make use of the Isabelle/Scala [97] library to generate HTML pages based on the proof structure, already containing the annotation of a page. Because the process has access to the full proof structure, it is easy to generate annotations: the main obstacle is that, at the time of these experiments, the information about the identifiers at this level did not distinguish clearly between declaration and use, so it is difficult to know what items to annotate with an `@about`.

4.6 Point-and-Write in Agora

We have implemented the mechanisms described in this chapter as part of the Agora prototype.[16] A current snapshot of the source can be found in our code repository.[17] Agora provides the following functionality, grouped by the tasks in the main use case. Writing and rendering the narrative is illustrated by the Agora page about the binomial coefficient.[18]

[16] `http://mws.cs.ru.nl/agora`
[17] `https://bitbucket.org/Carst/agora`
[18] `http://mws.cs.ru.nl/agora/cicm_sandbox/BinomialCoefficient`

Formalization. Agora allows the author to write her own formalizations grouped in *projects*, which resemble repositories of formal and informal documents. Agora has some support for verifying Coq formalizations, with an editor for changing the files. Alternatively, it allows an author to synchronize her working directory with the system (currently, write access to the server is required for this). Proof scripts from this directory are picked up, and provided as documents. Agora also scrapes Mizar's MML for HTML pages representing theories, and includes them in a separate project. Regardless of origin and editing methods, the proof scripts are rendered as HTML, and annotated using the vocabulary specified in Section 4.4.

Narratives. To allow the author to write natural language narratives, we provide the Creole Wiki syntax [76], which allows an author to use a lightweight markup syntax. Next to this markup and the antiquotation described next, the author can write formulae in LaTeX syntax, supported by the MathJax[19] library.

Reference. The author can include formal content from any annotated page by using the reference syntax, just giving page names to refer to pages within Agora, or referring to other projects or URLs by writing a reference of the form: `@{type location#name}`. For example, the formalization of the binomial coefficient in Coq is included in the Wiki, so it can be referred to by `@{oo:Definition CoqBinomialCoefficient#C}`. On the other hand, the Mizar definition is given at an external Web page. Because the URL is rather long, the reference is `@{oo:Definition mml:binom.html#D22}`. In this reference, `mml` is a prefix, defined using the Agora-specific command `@{prefix mml=http://mizar.cs.ualberta.ca/~mptp/7.12.02_4.178.1142/html}`. We do not consider prefixes a part of the "core" syntax, as another implementation could restrict the system to only work within a single Wiki, causing the links to be (reasonably) short.

Rendering a page written in this way, Agora transforms the Wiki syntax into HTML using a modified Creole parser. The modification takes the reference and produces a placeholder `div` containing the type, reference and repository of the reference as attributes. When the page is loaded, the placeholders are replaced asynchronously, by the referred-to entities. This step is necessary to prevent very long loading times on pages referring to many external pages. When the content is included, it is pre-processed to rewrite relative links to become absolute (with the source page used as the base URL), a matter of a simple library call.

An alternative to asynchronously fetching the referenced elements would be to cache them when the page is written. This approach could be combined with the asynchronous approach implemented, and would allow authors to refer to content that would, inevitably, disappear.

We currently have not implemented it, because it requires some consideration in the scope of Agora's storage model: the issue is whether Agora should store the included content as a backup, and how it should verify that the cached content is still up-to-date. Should the cache become invalidated, the system then needs a policy to decide whether and how to update the cache.

Appearance of Included Content. The appearance of included snippets depends on several things:

- Cascading Style Sheets (CSS) are used to apply styling to objects that have certain attributes. When including the snippets, they can either be styled using the information

[19] `http://www.mathjax.org`

in the source document (because the syntax is marked up according to rules for a specific system), or the styling can be specified in the including document (to make the rendered document look more uniform).

In the implementation, we statically include the CSS files from the source text. This is manageable due to the small number of included systems, with each system having its own pre-defined CSS style, that can me edited after inclusion to better fit Agora's style. A different approach would be to obtain the CSS styling from the source document, but this requires Agora to know about all styling directives that are applied to an included element, and potentially an interface that allows the author to override the 'default' styling.

- The system could use the included snippets to present the data using an alternative notation than the plain text that is typical for interactive theorem provers. This approach would require some system-dependent analysis of the included snippets, maybe going further than just HTML inclusion. The gathered data could then be used to render the included format in a new way, either specified by the author of the source text, or the author of the including text. This approach of "re-rendering" structured data was taken as part of the MoWGLI project [5], where the author of an informal text writes it as a *view* on a formal structure, including transformations from the formal text to (mathematical) notation.

  Because our approach intends to include a wide variety of HTML-based documents, we do not consider this notational transformation viable in general: it requires specific semantic information provided by the interactive theorem prover, which is not preserved in the annotation process described in Section 4.5, possibly not even exposed by the prover. However, where we have this information available, it would be good to use it to make a better looking rendered result.

4.7 Conclusions and Future Work

This chapter describes a mechanism for documenting formal proofs in an informal narrative. The narrative includes *pointers* to objects found in libraries of formalized mathematics, which have been *annotated* with appropriate types and names. The mechanism has been implemented as part of the Agora prototype. Our approach is Web-scalable in that the Agora system is independent from a particular formalized library: It may be installed in a different place, it references formal texts by URL, and it does not make any assumptions about the system underlying the library, except requiring an HTML+RDFa export. As future work, we see several opportunities for making the mechanisms more user friendly:

Include more systems. By including more systems, we increase the number of objects an author can refer to when writing a Wiki page. Because we made our annotation framework generic, this should not be a very difficult task, and single documents could be annotated by authors on the fly, if necessary.

Provide other methods for reference. At the moment, the `reference` part of our antiquotations is straightforward: the author should give a page and the identifier of the object in this page. It would be interesting to allow the author to use an (existing) query language to describe what item she is looking for, and use this to find objects in the annotated documents.

Improve editing facilities. Agora currently has a simple text box for editing its informal documents. We could provide some feedback to the author by showing a preview of the marked up text, including resolved antiquotations. More elaborately, we could provide an 'auto-completion' option, which shows the possible objects an author can refer to, limited by `type` and the partial path: if the author writes `@{oo:Theorem Foo#A}`, the system provides an auto-completion box showing all the theorems in the "Foo" namespace, starting with "A". This lookup could be realized in a generic way, abstracting from the different formalizations, by harvesting the RDFa annotations into an RDF database ("triple store") and implementing a query in SPARQL.

Consistency. The current design of the mechanisms already provides a better robustness than just including objects by giving a location, but can still be improved to deal with objects changing names. A solution would be to give objects an unchanging identifier and a human-readable name, and storing the antiquotation as a reference to this identifier. When an author edits a document containing an antiquotation, the name is looked up, and returned in the editable text.

Despite these shortcomings, we believe we have made significant steps towards a system in which authors can document formal mathematics by pointing and writing, without having to commit prematurely to a specific workflow, such as literate proving, or even a tool chain, because representations of (formalized) mathematics can be annotated after they have been generated.

Chapter 5

Authoring

This chapter is based on the paper "Proof in Context — Web Editing with Rich, Modeless Contextual Feedback" [81], with some minor expansions and references to other chapters in this work added.

5.1 Introduction

The Agora[1] system is a prototype for a "wiki for Formalized Mathematics" [28]: it provides web-based access to repositories of formal documents, allowing authors to write informal descriptions that include snippets of formal text [86].

Many formal documents are written in an iterative fashion: the author of a formal document writes commands for an interactive theorem prover, which interprets them to manipulate a "proof state": a list of assumptions and a goal that should follow from them. Based on this state, the author then writes new commands for the theorem prover, until the initial goal is dismissed as proven. Taken together, these commands form a proof *script*, which can be distributed to other users of the theorem prover. Because the script is meaningless without the proof states, a system that wants to give readers stand-alone access to the proofs should provide the proof states as well, a model which we have explored with the Proviola tool (Chapter 3).

For a wiki, it is not enough to just offer read-only access to the proofs: one of the main design principles for the first wiki was that *anyone* can edit *anything*, even if just by a little bit [95]. Additionally, reader understanding can be improved by allowing the reader to interact with the material in a "sandbox": an editor embedded in a document, that includes the material of the document for the reader to play with, for example to redo steps of the proof in a different way, or attempting to apply proven lemmas in different situations. For formal proof, there are two issues barring the way to an accessible editing experience:

Verification The appeal of a wiki for formal mathematics is that its (formal) content is verified by a proof assistant. Because a proof script rarely stands alone, but builds on other documents in a collection, each change to a document should lead to a re-verification of all the documents in the wiki, with respect to that change. This can take up a lot of time, that the user may not want to spend.

Interaction Because proof scripts are written interactively, a web-based editor should provide interaction. A standard wiki editor, on the other hand, is intended for writing simple

[1] `http://mws.cs.ru.nl/agora_ui`

markup and hyperlinks, which are tasks done in batch mode: the HTML is rendered after editing, instead of giving feedback to the author while editing. This is acceptable for the wiki syntax, which is readable by itself, and does not depend on state, but not enough for a programming language. In fact, the requirement for continuous feedback for programmers has recently gained an increased interest in the programming language community (see, for example, the language-oriented design by Burckhardt *et al.* [20]), since Brett Victor's "Inventing on Principle" talk [96].

Verification and External Consistency We leave the first issue for future work: Alama *et al.* [4] have presented some solutions to this issue on the system-and-tool level: their solutions focus on using build tools, such as GNU Make that allow a user to specify how a certain artefact should be compiled from its sources; dependency analysis and version control to provide more fine-grained control of theorem prover processes, so that the content that is verified is the absolute minimum and preventing any 'dirty' content, –content that forms a correct proof script, but removes certain lemmas that are necessary for other proofs–, from leaking into the wiki.

We believe that their solutions are not enough, as it forces user contributions to be evaluated in a strictly linear fashion, and always after the user committed to a single change. So the user needs to wait for a, possibly long-lasting, computation to finish, and cannot make a set of related changes, for example renaming a lemma at all places where it occurs. In addition to improving the tools to be more efficient in (re)verifying a collection of formal proofs, user interface support needs to be given that supports the authors in managing these tools: such interfaces can provide direct feedback on the user's changes with respect to the rest of the wiki (or a part thereof). Heuristic impact analysis, for example, can be used to display that a renamed lemma is used in several other pages of the wiki, or that it conflicts with another lemma. Such solutions prevent the user from committing a change that is bound to fail, and which the standard verification workflows would only detect when trying to apply the renamed lemma. To make the interface more responsive, full verifications need to move to the background, using advanced techniques such as those of Alama *et al.* Another improvement to the interface can be made by providing a user with a 'staging' area for gathering a group of related changes, that can be integrated at once into the wiki. Such a set of changes can then be verified as a whole and asynchronously, allowing batch changes and preventing user frustration. Both of the staging and the background tasks require an interface that give the author insight in the status of a group of changes. This requires giving a higher level view of interconnected theorems,[2] and a more significant setup than just an editor, however, and therefore is not addressed in this chapter.

This chapter describes a solution to the second issue: an editor that supports interaction with the content of the wiki, an editor that is mature enough to allow authors to edit existing documents, but accessible enough to be included as a sandbox for readers with little exposure to a theorem prover. To lower the threshold, we do not reuse the existing ProofWeb editor [49] or the Matita web editor [8]: these editors both use the "Proof General" [9] editing model which requires an extra action by the user, clicking a "proceed" button, before the theorem prover executes the commands written. This does not invite users to experiment, and can be, as we show here, entirely avoided.

Instead of using a lock-stepped model, we subscribe to an editor supported by *rich, modeless feedback* [25]: feedback to user changes that is given through a variety of means such as line

[2]Similar to the overviews given by modern programming language IDEs

highlighting and state windows (the rich part) and which is given as soon as possible, without forcing the user out of an editing “mode” (the modeless part). This model is similar to the document-oriented Isabelle/jEdit model described by Wenzel [97], but in a Web-based setting. The Clide interface [65] has a similar design, but inserts feedback inline, and focuses just on information computed by the theorem prover (this interface was developed in parallel with the interface described in this chapter). The Isabelle/jEdit model adds support for the Isabelle proof assistant to the jEdit editor: jEdit is a generic editor for programming languages, that can be expanded by writing plugins: pieces of (JVM-executable) code that react to changes in the edited document and user focus (communicated by cursor movements) in order to provide contextual information to the author. The Isabelle implementation uses these plugins to drive the theorem prover, and return contextual information to the author of a proof, such as a proof state and correctness information. The interaction is fully *asynchronous* and immediate: the prover calculates proof states based on the changes a user makes, and reports them back to the editor. In other words, the prover is ‘always on’, and does not require the author’s accord to continue computation.

The model of Isabelle/jEdit is preferable, as it does not force an author to synchronize prover and user focus: each time the user presses ‘proceed’, a new state is written, and used by the author to write a new step of the proof. While this is reasonable for a single proof, it does not allow the author to see the impact of changing the ‘design’ of a proof: changing a lemma can invalidate proofs that make use of the lemma, but this is only found out by stepping to the positions in which the lemma is used. In the asynchronous model, as much computation is carried out as is possible, something that is only possible because of recent improvements in processor speeds and amounts. Because computation is carried out on the entire proof text, the author gets immediate feedback on changes, even if the feedback occurs at the end of a document.

Introducing the asynchronous communication model to a web-based setting gives rise to a challenge and two advantages, compared to a stand-alone editor.

- The challenge is that because the client is served on a web page, while the theorem prover resides on a server, all communication is done through the HTTP protocol, which does not support the server pushing data. More modern techniques, such as WebSockets could be used to provide an easier model, but is not supported by all web servers and in particular not by the server configuration that Agora is currently deployed on. The solution is to have the editor continuously poll the server whether new data is available. Designing this protocol, that mimics the server pushing data is one contribution of this chapter. The major challenge is to design the protocol in such a way that the editor does not lose its reactivity. Techniques for server-push are being integrated into newer versions of web browsers, but the technique does not have a widespread library support: solutions still need to be hand-crafted, which costs as much effort as the design of the current protocol; .

- The first advantage is that because Agora’s editor is just a web page, displaying information is reduced to adding fragments of HTML to the page. This means that arbitrary server-side tools can work on the document, similar to the Isabelle/jEdit model, and that they can communicate their results as HTML fragments. On the contrary, tools that work in the Isabelle/jEdit model that want to report their results to the user would need to implement this display as a part of the jEdit plugin framework, which requires additional implementation work and a deeper knowledge of the jEdit environment. While this may in part be a matter of taste, and the benefit is negligible for more complex tasks, which might require server-side computation, adding new information to Agora’s editor view requires feedback as HTML (a markup language) instead of procedurally adding a viewing area to the editor and then connecting that area to the correct events.

- The second advantage is that Agora is web-based, and tools working with proofs can assume to have access to the Internet, and can use this fact to provide relevant information to the user, for example by showing similar formalizations in different theorem provers. An instance of Agora can be started offline, obviously disabling the Internet-based advantages, but still providing an editor for files in a repository.

In Section 5.2, we describe these advantages further, explaining what kind of tools can be attached to the editor, and how they can work with a proof. This section also includes a more in-depth look at what we intend the editor to support. In Sections 5.3 and 5.4, we describe how we overcome the challenge, by describing the document model as it exists on the server, in the client and in transit (Section 5.3) and how these incarnations are synchronized (Section 5.4) to hold the same data after one of the representations gets updated. In Section 5.5, we describe an important part of our implementation, a driver for the Coq [88] theorem prover. Because Coq currently does not support asynchronous computation, it needs to be 'faked' until the improvements by Wenzel, among others [99] become a stable part of Coq. This faking is done by adding and using extra information in the data structure, making Section 5.5 an example of how tools can enrich the data structure for their own purposes. Section 5.6 summarizes and gives a perspective on improving and using the editor.

5.2 Proving in Agora: Managing Context

Agora is a prototype wiki built upon existing repositories of formal mathematics: users can upload existing developments to the system and then collaborate on these documents in a Web-based system. This collaboration can include further development on the formal content, but the primary workflow of the system is based on writing informal pages describing the development, that can contain dynamic and interactive elements. If a document contains formal content, it is possible to bring up the theorem prover state on-demand for any line of formal proof (the Proviola system described in Chapter 3), and readers should be able to do exercises and experiments directly in the web interface.

Since content in Agora is imported from authors' existing formal development, we assume that there is an offline editor that supports more sophisticated workflows, and Agora's editor can focus on less involved, "one-off" editing tasks, in particular:

Edits during description An author describing a formal proof might discover improvements to this content. Changing the code to address these issues should not require the author to change back to the offline editor and resubmit the formal code: it should be possible to do edit the formal text inside the wiki's environment.

Exercises Several text books, most notably the Software Foundations text book [72], use a theorem prover to teach formal techniques in computer science. These text books are self-contained formal developments, that a student can run in a local theorem prover installation. Exercises in this text books take the form of formal proofs, which students need to complete. The benefit of a theorem prover is that a student gets direct feedback to whether or not a proof is correct, and that a teacher has a lighter load in verifying student assignments: because the theorem prover has verified the proofs, they will not contain factual mistakes, and a teacher can focus on improving a student's style. On the other hand, the documentation tools accompanying a theorem prover can be used to mark up a text book for online rendering, typically giving better results than the code highlighting in a theorem prover's offline editors. To combine the benefits of having an online theorem prover verifying student exercises with those of a text book as a rendered HTML page, it is necessary to supply students with an accessible editor that can be embedded in HTML documents.

Demonstrations If an author wants to demonstrate a formalization at a location where no theorem prover is available, having an editor at the same place as the formalization, which can easily load this formalization from the wiki can be useful for showing applications and alternatives.

These use cases are all covered by existing theorem prover environments on the web, such as ProofWeb, but we can lower the threshold further by making the editor modeless and generic. Having a generic editor is especially required in Agora, where we want to easily add new theorem provers, and also want to provide contextual feedback to the user, drawn from the wiki.

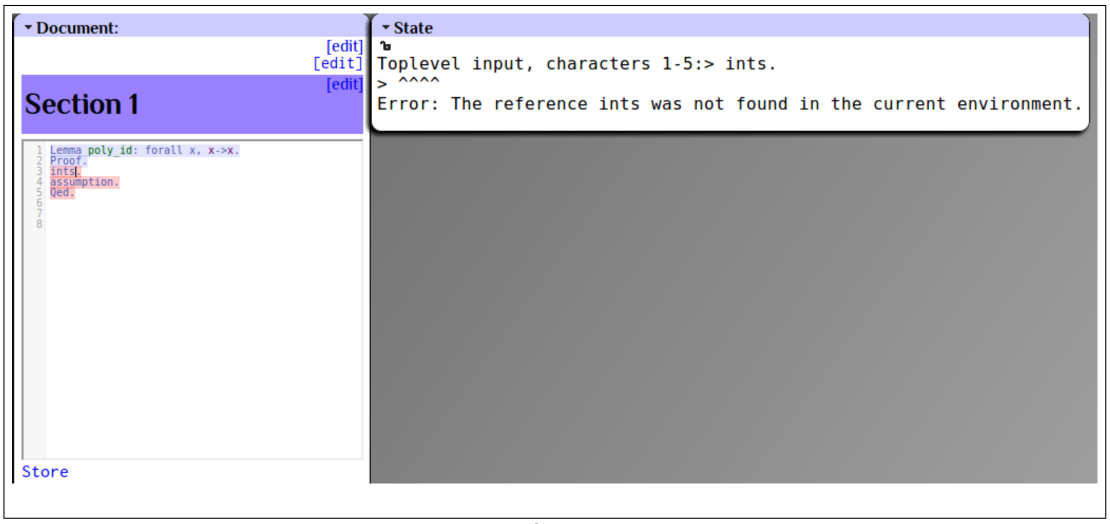
The advanced use cases concern proofs in the scope of a full-fledged formal development, where multiple proofs depend on each other, and an author can look into the history of a proof, and possibly use tools to analyse a proof while it is being edited. While Agora in theory supports these use cases through tool support, the system's architecture (and its implementation) are not robust enough to support them in practice. This is left to future work: the editor is a first step in providing authors with tools to write proofs in Agora, but we currently only consider the text of a formalization. Extensions can be made towards supporting the proof in the context of a larger formalization, or towards searching the wiki for appropriate lemmas, as well as integrating version control support.

Proof Context When an author writes a proof in Agora, the proof gets translated into a model, discussed in Section 5.3, that allows arbitrary tools to work on it: the tools add extra information to the commands the author writes. Because Agora is a web-based system, the tools reside on a server and can do intensive computations, possibly using the wiki or the entire Web to provide information. Furthermore, because the editor is already in a client-server model, the results are reported asynchronously, without disturbing the author. We call this model of editing *contextual* as the information displayed depends on the location of the text cursor: the information the author sees depends on the context in which it is being requested.. It is *rich*, using methods of communication beyond text-only dialogs, such as colouring and hyperlinks, and *modeless*, computing and reporting information while the author writes: the tools that provide the feedback add the information in the background, and the information is retrieved when the author shifts focus.

Figure 5.1 is a screenshot of Agora's editor in action. In this figure, the author has written a small proof for the Coq theorem prover. The following contextual information is computed by two server side tools:

State The state window shows an error, reporting that the command under the cursor is incorrect. This response was computed by the theorem prover.

Correctness The first two lines of the proof are correct, the others incorrect: Coq cannot recover from the error on line 3, as indicated by the colouring. The information for this colouring is computed by a post-processing step on the theorem prover output, which will be discussed in Section 5.5. The actual colouring is just a styling step using CSS on the client side.



```
Lemma poly_id: forall x, x->x.
Proof.
ints.
assumption.
Qed.
```

```
Toplevel input, characters 1-5:> ints.
> ^^^^
Error: The reference ints was not found in the current environment.
```

Figure 5.1: Contexts in action

Rich type information In the declaration of the lemma, its name (`poly_id`) and the bound variable (`x`) are coloured differently from the rest of the text. This information is obtained from the so-called "globalization" step in Coq's proof process: this step reports what types of identifiers (lemma, variable, . . .) occur at what locations. This information is computed by the theorem prover during evaluation and the location is reported as a character offset from the beginning of the file. Because the editor adds information to separate commands, a second tool normalizes the information to be a character offset from the beginning of the command.

Other tools can, for example, take the rich type information and evaluate references to lemmas and report them as hyperlinks to the place where they are defined. These hyperlinks can then be displayed as 'cheat sheet' window next to the editor.

The list above shows that the theorem prover process is just one of several processes working on a single model. However, the theorem prover has an elevated status: when the author writes a proof, the next command is typically determined by evaluating the state information the theorem prover reports: if the theorem prover reports and error, the author corrects it, and when the theorem prover reports a states in which an hypothesis has been simplified or a new goal is introduced, the author continues by writing a command that manipulates this new state. Traditional "Read-Eval-Print-Loop" models of theorem prover interaction are based on this driving principle: the author explicitly tells the theorem prover when to proceed on the proof, only proceeding as far as the focus of attention (the last command interpreted), even if the proof goes on beyond this focus. If the proof is changed not by appending new commands, but by changing existing commands, the changes might cause later steps to become invalidated, which is not indicated by a read-eval-print-loop process.

When contextual information is computed based on the proof for the entire proof script, the user no longer has to instruct the tools to start computing "on the next line". Instead, all tools that are available should start computing when new proof text becomes available, reporting their results asynchronously. We give a description of a theorem prover tool that fakes asynchronous updates in Section 5.5 as an example of a tool working on the server-side proof, but we first describe how the editor and server represent a proof.

5.3 Document representations

The client and the server share the proof's state in a proof *movie* [83], but each represents the model differently. This section describes the different models, including the model used to communicate changes from the server to the client. We describe the server model first: this serves as an introduction to proof movies.

5.3.1 Server model

The server stores the proof as a simplified proof movie: a list of *frames*. These frames contain *cells* that contain the information of the proof: each cell is a piece of information calculated by a tool based on other cells in the frame and, possibly, in previous frames. Each frame should contain at least a command cell. This cell holds theorem prover commands and comments, calculated by a light-weight parser.

Because we want to link the movie to the client's line-and-character-based model described below, we store the frames not in a list, but in a map from a line-character range to the frame containing the command occurring at that range. We assume that lookup occurs more often than modification, and store the map as a binary tree, indexed by line-character ranges. Because the ranges are contiguous, it is possible to look up a frame by a single text position: given a text position and a key, the position occurs either in the key, letting us return the frame, or it occurs before the key, leading to a descent in the left subtree, or it occurs after the key, leading to a descent in the right subtree.

Given a changed portion of text, we can find the nodes that are affected by this change, and recompute a new tree from this change by taking the affected frames and recomputing new frames from the changed text. After inserting the changed frames, the tree looks like "preceding frames + changed + following frames", where the "preceding" and "following" frames are the old frames that surround the changed portion. Keeping the old frames, especially the preceding ones, is an advantage when feeding the frames to a theorem prover, as we describe in Section 5.5. In short, the data used for driving the proof assistant is kept in the old frames, and can be used to prevent unnecessarily computing with these frames.

5.3.2 Client model

The client model is restricted by the representation of the data structure in HTML: while it is possible to manipulate a data structure as rich as the original movie in JavaScript, it will eventually need to be shown to, and interacted with by, the author in an HTML document. Because of this, and to minimize redundancy of the implementation over several languages, we keep a minimal model in JavaScript, containing only the information we want to show in the client.

In the client, the author needs to have at least the text of the proof, and feedback on how the proof is processed: in terms of the movie, the client needs the commands and proof states for each frame. The commands are placed in some editable text area, the states are requested in response to text cursor movements: following the Isabelle/jEdit model, we take the text cursor position to be indicative of the author's focus of attention, and therefore guiding in what state to show. This heuristic is not completely correct, as is shown by an e-mail thread started by Nipkow in the Isabelle users' mailing list[3]: when extending the proof based on the current state, the text cursor moves while typing, causing the system to request the new, possibly non-existing and most likely erroneous, state. On the other hand, when the author

[3]`https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2012-July/msg00182.html`

has finished typing a command and it is still erroneous, the state of the last correct command can be useful in debugging. More experiments and prototyping is necessary to see how to properly deal with this scenario.

We have a choice in how to represent the text in HTML: we can use a `textarea` element, a standard HTML DOM element with the `contenteditable`[4] attribute enabled, or an "off-the-shelf" editor that is programmable to work with our data.

Textarea Textarea elements hold plain text and respond to different user editing actions: typing, copying and pasting. Its content cannot be marked up, which makes it not suitable for rich modeless feedback. On the positive side, it does not apply formatting to the text being written, allowing it to be gathered as plain text, which is what the theorem prover expects.

Contenteditable Using contenteditable on the other hand, causes HTML markup elements to be inserted in non-standard ways, requiring a cleanup of the text before it is processed further. As described by others [43, 49], this cleanup is non-trivial to implement, and would require effort to be kept synchronized with the models different browsers use. Elements with `contenteditable` do have the advantage of being just HTML elements, exposed to JavaScript manipulation and CSS styling.

Off-the-shelf editor A third option is to use a third-party editor component, which allows text to be obtained as plain text, while also providing markup facilities through a programming interface. A feature-rich and easy-to-use representative of this family is CodeMirror[5], which is also used for several high-profile projects. We have chosen this editor in order to combine the ease-of-use of a `textarea` element, while also being able to markup the code the author writes to provide feedback.

An additional benefit of CodeMirror is that it allows a *mark* to be set for a region of text. While the intention of these marks is to apply custom styling to arbitrary regions of text, the fact that each mark is a JavaScript object, and that a JavaScript object can have new attributes added at runtime, allows us to store arbitrary data to regions of text, corresponding to the information found in each frame. This gives us a way to represent the movie in the editor: a frame can be represented by creating a mark on the text in its command cell, and other cells can be added as attributes to this mark. When the text cursor is moved, the mark under the cursor can be retrieved, and its data can be used to render the context.

5.3.3 Communication

For communication, the movie is represented as a list of frames in the "JavaScript Object Notation", a textual representation of JavaScript objects, including lists and strings. Instead of sending over the entire frame, a frame is represented by an object, containing the text range its command covers, split in a `start_line` and a `start_char`, and an `end_line` and an `end_char`. This suffices, because the client already knows the text content. The rest of the data per frame depends on the computations carried out on the movie, such as those described in Section 5.5: each cell is provided by some tool, with tools providing one or more cells.

[4] `http://www.w3.org/TR/2008/WD-html5-20080610/editing.html#contenteditable0`
[5] `http://codemirror.net`

5.4 Synchronizing the document

When the author updates the text in the editor, this should update the document's movie. In turn, this triggers the server-side processes that compute on the new movie. The results of these computations need to be communicated back to the server. Because the editor should remain reactive when this computation is in progress, and computation takes time, the computation is executed asynchronously and the results are reported separately. During computation, the author is free to edit the text, restarting the process. This leads us to distribute the synchronization over two protocols:

1. An UPDATE_SERVER protocol which takes an update of the client text and updates the server model, triggering the start of computation.
2. An UPDATE_CLIENT protocol which obtains the tool updates to the server model and communicates them to the client.

These two protocols execute asynchronously: UPDATE_SERVER is executed every time the user updates the client model, and UPDATE_CLIENT runs while there is new information to provide to the client, its effect is that the server pushes new information to the client.

From the perspective of the server, there is not much difference between the client's updates to the movie and the updates from other processes, so both protocols might be merged in a single protocol that takes an arbitrary change to a movie and broadcasts it to all other processes. We prioritize the UPDATE_CLIENT protocol for two reasons: first, all computation on the movie is based on the commands from the client, so the other processes are started when new commands come in; second, because the client's only way of communication is through HTTP, which does not support server-side pushes on all browsers, its update protocol requires some extra care.

The figures in the next two sections represent all processes working on the movie as a single, anonymous process i. Messages sent to this process should be seen as being sent to all processes.

5.4.1 UPDATE_SERVER: updating the server-side from the client

UPDATE_SERVER, depicted in Figure 5.2, is a straightforward protocol: it is initiated whenever the client's content is updated with new text. It starts by the client sending this text to the server. In reaction to receiving this text, the server acknowledges this reception, allowing the client to remain responsive. This information, including what commands are scheduled for computation, is updated by the UPDATE_CLIENT protocol, which should execute as soon as possible after UPDATE_SERVER.

The server then creates a movie based on the old movie, m, and new text, t', using a *camera* function. This function takes the following steps:

1. Get the commands from m as a single text t.
2. Compute the difference $\delta(t, t')$. This difference is a list of "patch" operations and should mention at which locations in t changes occur.
3. Using the locations, find the frames in m that need to be changed, f_c, as well as the frames following them: f_n. Get the text fragments from the frames f_c and apply the patch to this text, obtaining a new fragment f'.
4. Parse f' into a list of frames.

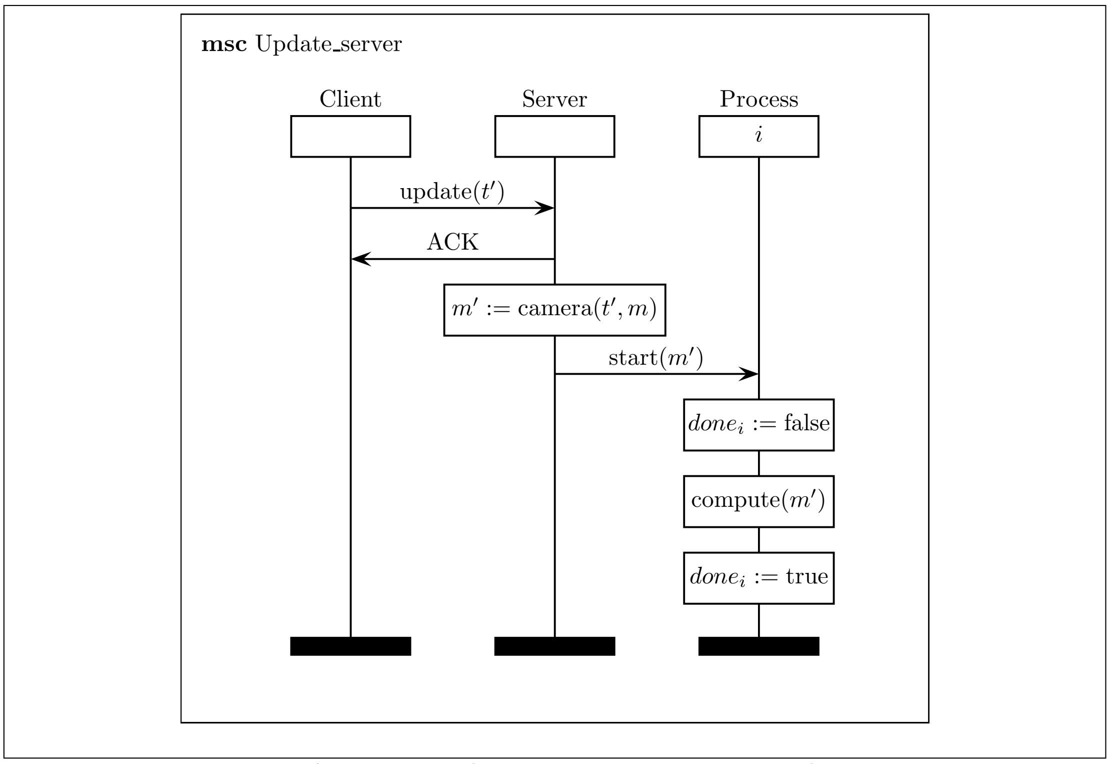


Figure 5.2: The UPDATE_SERVER protocol

5. Insert the new frames and reinsert the frames from f_n to make sure all frames are indexed by their new textual location, clear all tool information in f_n, this invalidates all f_n, so they will be re-processed.

The parsing in step 4 transforms the text into a list of commands and comments, based on the theorem prover's syntax. This is a naive scanner for command terminators described previously for the Proviola tool (Chapter 3), and similar to the *read* phase described by Wenzel [98].

Regarding step 5: one could only update the keys of the frames from f_n instead of reinserting the frames. In our implementation, we currently do not follow this model for ease of implementation: it would require determining what keys have been replaced and using this to compute an offset to be applied to all the keys for f_n.

After the new tree is computed, all tools subscribed to it are notified, and given a pointer to the new document. These processes can then compute with the new information. After a process has finished computing, it sets a *done* flag, allowing the server to poll for finished computation in the data retrieval protocol UPDATE_CLIENT, which is described in the next section.

5.4.2 UPDATE_CLIENT: getting frame data from the server

At the end of the UPDATE_SERVER protocol, the server started a number of processes, which work on the content in the movie. During the computation, a process can update frames in the movie with their own data. The goal of the UPDATE_CLIENT protocol is to send this data to the client.

In a normal client-server situation, the server would just push the information to the client, but the client and server communicate over HTTP, which only allows a server to respond to client request. This means we need to mimic server push using JavaScript, using a technique inspired by the *Comet* protocol [39].

Comet is a technique that emulates server push through a long-standing, asynchronous request: the client requests data from the server, which then waits until it has data to send to the client. The server then responds to the request with this data. This response is delivered through a callback function to the client, meaning the client can continue after requesting data. In the callback function, the client processes the data, and immediately starts a new request. If there is no data to send, the request is held indefinitely, as long as the implementation allows. Should the request time out, the client can request it again.

A normal Comet implementation assumes that the data on the server side comes in continuously, from a fixed data source. A typical example would be a 'ping' command executed by the server, with the results sent to the client. This means that each time the client posts a new request, the server can just read the output from the command, and send it on.

In our case, the information in the movie comes in in bursts: the process, such as a theorem prover, can take quite a long while before it has finished computing the information for a single frame, but then quickly fill the next three frames. When this information comes in, the server needs to push it as soon as possible. Another difference from the traditional Comet setup is the fact that the results of the computations are stored directly in a shared data structure between the server and the processes (the movie), instead of being streamed directly, using some sort of callback function. In the case of a callback, the server can reply to the client as part of responding to the callback function, but in the case of a shared data structure, the server retrieves the data structure and sends it to the client, after determining if it has changed. A final piece of the protocol is the fact that processes can be finished; afterwards, they do not produce new data, meaning the server can use the data it gathered most recently. To allow the server to query tools for new data, we equip each tool with a "get_data" function: a function that, given a frame, reports the data it has computed for that frame. The server can use this function for each frame until the process has stopped.

We design the UPDATE_CLIENT protocol using the following constraints:

- The client initiates the protocol.
- The server sends as much data from the processes as is available.
- The data is sent as soon as possible.
- The server only sends if there is data.

The data is sent as soon as possible to allow the client to respond to the author's editing as soon as possible, giving feedback when it is available, instead of waiting for a potentially long-running computation to finish.This allows the author to react to early problems, without waiting for a failing computation that might be caused by this earlier error.

The last item is meant to minimize the number of transactions between client and server, to improve the responsiveness of the client. We do not worry that some data might be duplicated among messages, because we expect these messages to be reasonably small.

The full protocol is shown in Figure 5.3.

This protocol starts the moment the client asks for the movie through the 'update_client' message. If all processes are finished, the server will hold this request, until at least one of the processes starts again, as a result of the UPDATE_SERVER protocol. When there exist processes which are not finished, the server obtains data from these processes for each frame

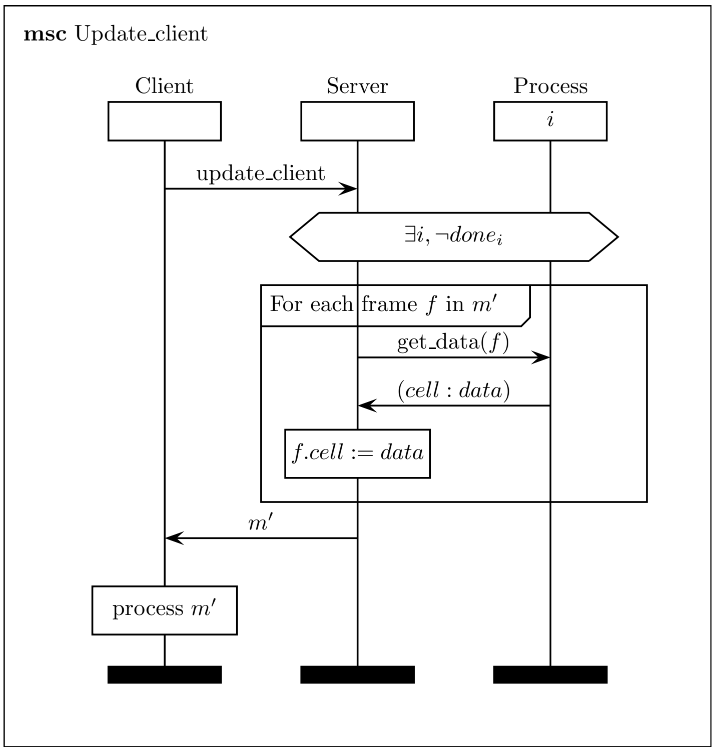


Figure 5.3: The UPDATE_CLIENT protocol

f in the movie m'. The processes return this data, indexed by the cell in which this data belongs. The server then stores the data in the corresponding cell of f. After a full pass over the movie in this manner, the new movie is converted to JSON and sent to the client. The client will immediately restart the protocol and process the data.

This protocol satisfies our constraints:

- The client initiates the protocol through the update_client message.
- By getting data from all processes for each frame, the server sends as much data as is available, assuming the processes report all information computed thus far.
- The server updates the movie it sends to client immediately after a request is made: it does not wait for computations to finish.
- The server holds the last request of the client, if there are no processes running.

During the process step, the client uses the data in the movie to add contextual information to the editor text and arrange the information to appear when the text cursor moves, as described in Section 5.2.

5.5 Computing with the documents

We have now described the data structures used by client and server and the protocol to synchronize changes in these data structures, but have not yet described the implementation of the processes. There are no requirements on these implementations, apart from reporting data to the server as described in Section 5.4.2. In this section, we describe an implementation of a theorem prover process as an example of the processes that start when the server-side movie is changed.

Processes are connected to changes in the movie following the Observer pattern [33]: each time the movie updates, the processes are started with the new set of frames. These frames are the only data the processes share with each other and the server. To cache data, the processes are allowed to write to specific cells of each frames. This prevents individual threads of conflicting over the frames and overwriting each other's data, provided the assigned cells are disjoint.

Having processes produce data in pre-specified cells allows the system to schedule processes that depend on data produced by other processes: each process declares what cells they depend on and what cells they will fill and the scheduler will make sure that once a cell is filled, the interested parties get notified to start their computation. We will give a concrete example later in this section.

5.5.1 A theorem prover driver

As an example of a more involved process, we describe how we can drive the Coq theorem prover, based on changes in the movie, by storing enough data in frame cells to make the theorem prover 'fake' the behaviour of Isabelle's asynchronous processing [98]: while work is underway in making Coq's interaction model asynchronous [15], Coq does not yet support this model, and we build a driver around the existing, lock-step, model that allows the server to update the entire movie, and gets states back as efficiently as possible. We do not implement this as a modification for Coq, because this requires working with the internal (programmer's) interfaces of the system, which are more likely to change than the external (user) interfaces. Even if the internal interfaces were stable, the state management lies deeply embedded in

Coq's source code, requiring many changes to get the model we now obtain by a fairly simple protocol.

Before we can describe the driver, we need to explore what frame cells we will have it fill. To do this, we look at how Coq processes a proof script.

State management and interpretation order

As described before, an interactive theorem prover takes commands and returns responses. This is the first cell our process will fill. Responses cannot be calculated directly from a single command: instead, a response is computed from a context, consisting of at least a proof state, but which could also including automation hints, notational conventions and other side effects. This means that there is an interpretation order of the frames in a movie: in order to interpret frame f_2, the theorem prover first needs to have processed frame f_1. In modern models of the theorem prover, the dependencies between commands are not seen as a list, but as a directed, acyclic graph, the nodes of which can be interpreted asynchronously (or lazily), provided all previous nodes have been evaluated. In theory, we support both models: a frame f holds a list of 'dependencies': the frames that need to be evaluated before f gets evaluated. In our implementation, the server assigns these dependencies in a linear fashion, so each frame depends on exactly one other frame, the one immediately preceding it in the proof, but this function can be replaced by a process doing DAG analysis. This would require our driver to depend on a cell holding the dependencies, and to have some way of scheduling frames for computation. This scheduling can be done, depending on the theorem prover implementation, either by linearizing the graph, or by submitting frames in parallel.

When the command of a frame changes, the driver can use the dependencies of the frame to determine which frames it needs to send to the theorem prover. Naively, it can send all the dependencies from scratch. Less naively, the driver can keep a theorem prover online, and use certain, system-specific commands to 'undo' previous commands, in order to get to the required context.

Coq's state management For Coq, we can obtain some state management by keeping extra administration: for each command, we document which "state number" the theorem prover emitted after executing it. The state number reflects the number of commands Coq has executed successfully since it has been started and is reported through the program's prompt, provided it is given the `-emacs` switch. This state number, n can be provided to the `BackTo` command, causing the system to backtrack to a state m, such that $m \leq n$. The caveat for this method is that it is possible to 'overshoot' the correct state number: the system is unable to jump back into the proof of a lemma, and will instead skip to the state before the lemma was stated. This process is illustrated in Figures 5.4 and 5.5, both figures are obtained using Coq 8.4's `coqtop` tool, started with `coqtop -emacs`. The responses are replaced by ellipses for the sake of brevity. Additionally, the warning in Figure 5.5 contains non-printing ASCII characters that have been removed before typesetting.

In Figure 5.4, the `BackTo` is given while within the proof of the lemma. This brings the proof state exactly back to the requested number, as evidenced by the last prompt. In Figure 5.5, on the other hand, the `BackTo` command is given when the proof is closed. Instead of jumping back into the proof and reverting to state 2, the theorem prover returns to the state before the lemma was state, warning the user that the system is not in the exact state requested.

```
Welcome to Coq 8.4 (October 2012)

<prompt>Coq < 1 || 0 < </prompt>Lemma foo: forall x, x→x.
...
<prompt>foo < 2 |foo| 1 < </prompt>intros.
...
<prompt>foo < 3 |foo| 2 < </prompt>BackTo 2.
...
<prompt>foo < 2 |foo| 1 < </prompt>
```

Figure 5.4: `BackTo` within a proof

```
Welcome to Coq 8.4 (October 2012)

<prompt>Coq < 1 || 0 < </prompt>Lemma foo: forall x, x→x.
...
<prompt>foo < 2 |foo| 1 < </prompt>intros.
...
<prompt>foo < 3 |foo| 2 < </prompt>Admitted.
...
<prompt>foo < 4 |foo| 2 < </prompt>BackTo 2.
Warning: Actually back to state 1.

<prompt>Coq < 1 || 0 < </prompt>
```

Figure 5.5: `BackTo` outside of proof

Driving Coq

To make use of the state mechanism in Coq, we use the following system. In this description, we assume that frames contain extra data in *cells*. This data is used to determine what commands to send to the Coq process a summary of the cells is described after the system description:

Initialization Initially, all frames contain a "reached state" cell. This cell is initialized to the value "undefined". Following the order recorded in the "dependencies" cell, the commands of the frames are submitted to the theorem prover. After each frame, the response and the reached state are recorded in the appropriate cells.

On change When a frame f changes:

1. Update its reached state cell to contain the undefined value. Also set the reached state for all following frames, with respect to the order, to undefined: because f sets up a context that the following frames make use of, it is necessary to recompute these frames in the new context.

   It is possible that the dependency DAG changes as a result of f changing, causing some of the frames following f to be "dangling" (no longer dependent on any frame). Dealing with this situation is beyond the scope of this chapter; a solution would include quickly determining the changes in the DAG that were caused by the change in f, and only scheduling those frames that occur in the new DAG.
2. For all dependencies of f, find the dependency with the highest reached state, n.
3. Send `BackTo n.` to Coq, record the actually reached state, m: as described in the previous paragraph, Coq might go to a state number that is less than n.

4. For all frames f' such that the reached state of f' is between m and n, send the command of f' to Coq. Coq is now in state n.
5. For each frame that still has an undefined reached state, send that frame's command (with respect to the DAG), and record the reached state after each command.

This protocol stores extra information in the frames: the cell "reached state" contains the state number that the theorem prover reported after executing the frame's command. It is used to revert to a relevant state before executing a changed command, as illustrated above. This state can be used for a different purpose, namely giving feedback about the correctness of a command: whenever a command gets executed successfully, the state counter will increase by one. When a command results in an error, the state is no longer increased. We record this correctness information in its own cell.

In summary, the Coq process only requires command cells, and optionally dependency cells. It fills response, state and correctness cells. The response and correctness cells are used by the client, while the state cell is reused by the process itself, to manage the state of the theorem prover.

Expanding to different tools

We can also use the Coq process to generate globalization information, as described in Section 5.2. We will not detail this process here, as the data format Coq uses to report the globalization is rather technical. The information contained in the globalization gives the exact (*global*) location of any formal elements within a group of files, as well as the full syntactical type of these elements. When this globalization information is available, a second tool can require the globalization, and generate hyperlinks from this information, in a similar way as Coq's HTML generator, `coqdoc`. In our framework, this is a process that requires a globalization cell and produces a hyperlink cell. This cell can be used by the client to display the hyperlinks next to the editor, or update the editable text to contain the hyperlink. The latter does pose a usability question, as clicking on an editable hyperlink can either be taken as a navigational action (following the link) or as a focussing action (placing the cursor to change the hyperlinked text): it should be clear to the user what the result of clicking in the editor is, and how she can change the intended meaning of her click. For example, typical feedback for "you are going to follow a link" is changing the cursor to a 'pointing finger', and the author can modify this behaviour by holding a key while clicking: this changes the cursor to a text selection cursor, an allows the user to select the hyper-linked text. For an editor, the converse might be a better default, where holding a key when clicking allows the author to follow a link.

5.6 Conclusions and Further Work

This chapter has presented a method of using a *generic* web editor component to communicate with a theorem prover, by using a shared data structure, without making assumptions about the theorem prover. In fact, the theorem prover is seen as a generic process working on the data structure, whose results get reported to the client. A prototype implementation can be found at `http://mws.cs.ru.nl/agora_ui/`.

Having this editor available, we can consider the following expansions:

Expand to other theorem provers In this chapter, we have only implemented communication with the Coq theorem prover, but it should be possible to adapt other systems to work with the same protocol. In particular, the Isabelle theorem prover, through its

Scala interface, already has a model similar to our movies, so it should be easy to convert between the two models. In Chapter 6, we give an implementation for the HOL Light theorem prover for this model.

More secondary processes There are more processes that take a representation of a proof and provide some information about that proof. One example is the proof adviser service for the Mizar theorem prover [92], which provides references to lemmas that will help to prove the current subgoal.

Embed in documents We have not yet embedded our editor in documents in the wiki, but it would be interesting to post-process, for example, the Software Foundations notes [72] to include editors for the example. We imagine these documents to look similar to the interactive tutorial for the CoffeeScript language that can be found online at http://autotelicum.github.com/Smooth-CoffeeScript/.

Use in wiki workflow Finally, the proof editor has a place in the writing process of Agora: for this, we need to investigate how to best store documents in a wiki for formal mathematics, and gear the editor to fit in this workflow. In particular, we might want to include tools that provide version control information about the proof documents in the wiki, as well as information on the impact that changing (part of) a proof has on other documents in the wiki.

Embedding the editor in actual workflows will undoubtedly reveal problems both in implementation and design, but the current prototype shows that it is viable to have a rich, modeless editor for formal proof, that works in a web based setting, lowering the threshold considerably for non-specialists to enter the field.

Chapter 6

Case Study

This chapter describes the use of Agora in publishing the description of a real formal development, supplemented by the formalization. The formalization chosen is (a chapter of) the Flyspeck project [40], which formalizes the proof the Kepler conjecture. This chapter is based on the paper "Communicating Formal Proof: The Case of Flyspeck" [84] by Tankink, Urban, Kaliszyk and Geuvers, and the extended version of that paper, "Formal Mathematics on Display: A Wiki for Flyspeck" [85], by the same authors.

6.1 Introduction

The formal development of (large) mathematical theories is gradually becoming a mature field of mathematics and computer science. In various proof assistants, large repositories of formal proof have been created, e.g. in Mizar [38], Coq [16], Isabelle [55] and HOL Light [42]. This has led to fully formalized proofs of some impressive results, for example the odd order theorem in Coq [37], the proof of the four colour theorem in Coq [36] and a significant portion of the proof of the Kepler conjecture [41] in HOL Light.

Even though these results are impressive, it is still quite hard to get a considerable speed-up in the formalization process. If we look at Wikipedia, we observe that due to its distributed nature everyone can and wants to contribute, thus generating a gigantic increase of volume. If we look at the large formalization projects, we see that they are very hierarchically structured. This holds despite the fact that proof assistants support a distributed way of working without a strict hierarchy of who can change what, and that version control systems make it easy to back out of an unwanted change. An important reason is that the *precise* definitions *do* matter in a computer formalised mathematical theory: some definitions work better than others and the structure of the library impacts the way you work with it.

There are other reasons why formalization is progressing at a much slower rate than, e.g. Wikipedia. One important reason is that it is very hard to get access to a library of formalised mathematics and to reuse it: specific features and notational choices matter a lot and the library consists of such an enormous amount of detailed formal code that it is hard to understand the purpose and use of its ingredients. A formal repository consists of computer code (in the proof assistant's language), and has the same challenges as source code in mainstream programming languages regarding understanding, modularity and documentation. Also, if you want to make a contribution to a library of formalized mathematics, it has to be completely verified until the final proof step. And finally, giving formal proofs in a proof assistant is very laborious, requiring a significant amount of training and experience to do effectively.

To remedy this situation we have been developing the Agora platform: a wiki-baed system that supports the development of large coherent repositories of formalised mathematics. We illustrate our work by focusing on the case of a wiki for the Flyspeck project, but the aims of Agora are wider. In short we want to provide proof assistant users with the tools to

1. document and display their developments for others to be read and studied,
2. cooperate on formalizations,
3. speed up the proving by giving them special proof support via AI/ATP tools.

All this is integrated in one web-based framework, which aims at being a "Wiki for Formal Mathematics". In the present chapter we highlight and advocate our framework by showing the prototype Flyspeck Wiki. We first elaborate on the three points mentioned above and indicate how we support these in Agora.

Documenting formal proofs An important challenge is the communication of large formalizations to the various different communities interested in such formalizations: PA users who want to cooperate or want to build further on the development, interested readers who want to understand the precise choices made in the formalization and mathematicians who want to convince themselves that it is really the proper theorem that has been proven. All these communities have their own views on a formalization and the process of creating formalization, giving a diverse input that benefits the field. Nonetheless, communicating a formal proof is hard, just as hard as communicating a computer program.

Agora provides a wiki based approach: Formal proofs are basically program code in a high-level programming language, which needs to be documented to be understandable and maintainable. A proof development of mathematics is special, because there typically is documentation in the form of a mathematical text (a book or an article) that describes the mathematics informally. This is what we call the *informal mathematics* as opposed to the *formal mathematics* which is the mathematics as it lives inside a proof assistant. For software verification efforts, there is no pre-existing documentation, but Agora can be used to provide documentation of the verification as well. These days, informal mathematics consists of LaTeX files and formal mathematics usually consist of a set of text files that are given as input to a proof assistant to be checked for correctness. In some cases, the formal mathematics also contains informal descriptions in some markup language, but this is tied to a particular way of working.

In Agora, one can automatically generate HTML files from formal proof developments, where we maintain all linking that is inherently available in the formal development. Also, one can automatically generate files in wiki syntax from a set of LaTeX files. These wiki files can also be rendered as HTML, maintaining the linking inside the LaTeX files, but more importantly, also the linking with the formal proof development. Starting from the other end, one can write a wiki document about mathematics and include snippets of formal proof text via an inclusion mechanism. This allows the dynamic insertion of pieces of formal proof, by referencing the formal object in a repository.

Cooperation on formal proofs With Agora, we also want to lower the threshold for participating in formalization projects by providing an easy-to-use web interface to a proof assistant [81]. This allows people to cooperate on a project, the files of which are stored on the server.

Proof Support We provide additional tools for users of proof assistants, like automated proof advice [51]. The proof states resulting from editing HOL Light code in Agora are continuously sent to an online AI/ATP service which is trained in a number of ways on the whole Flyspeck corpus. The service automatically tries to discharge the proof states by using (currently 28) different proof search methods in parallel, and if successful, it attempts to create the corresponding code reconstructing such proofs in the user's HOL Light session.

To summarize, the Agora system now provides the following tooling for HOL Light and Flyspeck:

- a rendering of the informal proof texts, written originaly in LaTeX,
- a hyperlinked, marked up version of the HOL Light and Flyspeck source code, augmented with the information about the proof state after each proof step
- transclusion of snippets of the hyperlinked formal code into the informal text whenever useful
- cross-linking between the informal and formal text based on custom Flyspeck annotations
- an editor to experiment with the sources of the proof by dropping down to HOL Light and doing a formal proof,
- integrated access to a proof advisor for HOL Light that helps (particularly novices) to finish their code while they are writing it, or provide options for improvement, by suggesting lemmas that will solve smaller steps in one go.

Most of these tools are prototypical and occasionally behave in unexpected ways. The wiki pages for Flyspeck can be found at `http://mws.cs.ru.nl/agora_cicm/flyspeck`. These pages also list the current status of the tooling.

The rest of the chapter is structured as follows. Section 6.2 shows the presentation side of Agora, as experienced by readers. The internal document model of Agora is described in Section 6.3, Section 6.4 explains the interaction with the formal HOL Light code, and Section 6.5.1 describes the inclusion of the informal Flyspeck texts in Agora. Section 6.6 concludes and discusses future work.

6.1.1 Similar Systems

There are some systems that support combining informal documentation with computed information. In particular, Agora shares some similarities with tools using the OMDoc [59] format, as well as the IPython [71] architecture (and Sage [80], which uses IPython as an interface to computer algebra functionality).

OMDoc is mainly a mechanization format, but supports workflows that are similar to Agora's. However, it differs in execution: OMDoc is a stricter format, requiring documents to be more structured and detailed. In particular, this requires its input languages, such as sTeX, to be more structured. This structuring forces the author to manually point out how concepts used in the informal narrative map to semantic concepts. On the other hand, Agora does not define much structure on the files its includes, rather extracting as much information as possible and fitting it in a generic tree structure. Because Agora is less strict in its assumptions, it becomes easier to write informal text, freeing the authors of having to write semantic macros. Agora could output documents that are structured in OMDoc based on the informal input, but because our implementation has not yet stabilized, we have not committed to any external data format: it is easier to adapt our own formats than try and work within the restrictions of an existing framework.

The IPython architecture has the concept of a *notebook* which is similar to a page in Agora: it is a web page that allows an author to specify 'islands' of Python that are executed on the server, with the results displayed in the notebook. Agora is based on similar principles, but extends upon them by having a collection of documents referring to each other, instead of only allowing the author of a document to define new islands.

6.2 Presenting Formal and Informal Mathematics in Agora

Agora has two kinds of pages: fully *formal* pages, generated from the sources of the development, and *informal* pages, which include both markup and snippets of formal text. To give readers, in particular readers not used to reading the syntax of a proof assistant, insight in a formal development, we believe that it is not enough to mark up the formal text prettily:

- there is little to no context for an inexperienced reader to quickly understand what is being formalized and how: items might be named differently, and in a proof script, all used lemmas are presented with equal weight. This makes it difficult for a reader to single out what is used for what purpose;

- typically, the level of detail that is used to guide the proof assistant in its verification of a proof is too high for a reader to understand the essence of that proof: it is typically decorated with commands that are administrative in nature, proof steps such as applying a transitivity rule. A reader makes these steps implicitly when reading an informal proof, but they must be spelled out for a formal system. In the extreme, this means that a proof that is 'trivial' in an informal text still requires a few lines of formal code;

- because most proof assistants are programmable, a proof in proof assistant syntax can have a different structure than its informal counterpart: proofs can be 'packed' by applying proof rules conditionally, or applying a proof rule to multiple similar (but not identical) cases.

On the other hand, it is not enough to just give informal text presenting a formalization: without pointers to the location of a proof in the formal development, it is easy for a reader to get lost in the large amount of code. To allow easier navigation by a reader, the informal text should provide *references* to the formal text at the least, and preferably include the portions of formal text that are related to important parts of the informal discussion.

By providing the informal documentation and formal code on a single web platform, we simplify the task of cross-linking informal description to formal text. The formal text is automatically cross-linked, and annotated with proper anchors that can also be referenced from an informal text. Moreover, our system uses this mechanism to provide a second type of cross-reference, which includes a formal entity in an informal text ([86], Chapter 4 of this thesis): these references are written like hyperlinks, using a slightly different syntax indicating that an inclusion will be generated. Normal hyperlinks can refer to concepts on the same page, the same repository, or on external pages.

These mechanisms allow an author of an informal text to provide an overview of a formal development that, at the highest level, can give the reader insight in the development and the choices made. Should the reader be interested in more details of the formalization, cross-linking allows further investigation: clicking on links opens the either informal concepts or shows the definition of a formal concept.

The formalization of the Kepler conjecture in the Flyspeck project provides us with an opportunity to display these techniques: it is a significant non-trivial formalization, and its informal description in LaTeX [40] contains explicit connections between the informal mathematics and the related formal concepts in the development. We have transformed these sources into the wiki pages available on our Agora system[1]. Parts of one page are shown in Figures 6.1 and 6.2.

6.2.1 Informal Descriptions

The informal text on the page is displayed similarly to the source (Flyspeck) document, from which it is actually generated (see Section 6.5.2), keeping the formulae intact to be rendered by the MathJax[2] JavaScript library. The difference to the Flyspeck source document is that the source document contains *references* to formal items (see also Section 6.5.1), while the Agora version *includes* the actual text of these formal entities. To prevent the reader from being confused by the formal text, which can be quite long, the formal text is hidden behind a clearly-labeled link (for example the `FAN` and `XOHLED` links in Figure 6.1 which link to the formal definition of *fan* and the formal statement of lemma *fan_ cyclic*).

The informal page may additionally *embed* editable pieces of formal code (instead of just including addressable formal entities from other files as done in the demo page). In that case (see Section 6.4) clicking the 'edit' on these blocks opens up an editor on the page itself, which gives direct feedback by calling HOL Light in the background, and displaying the resulting proof assistant state, together with a *proof advice* which uses automated reasoning tools to try to find a solution to the current goal.

6.2.2 Formal Texts

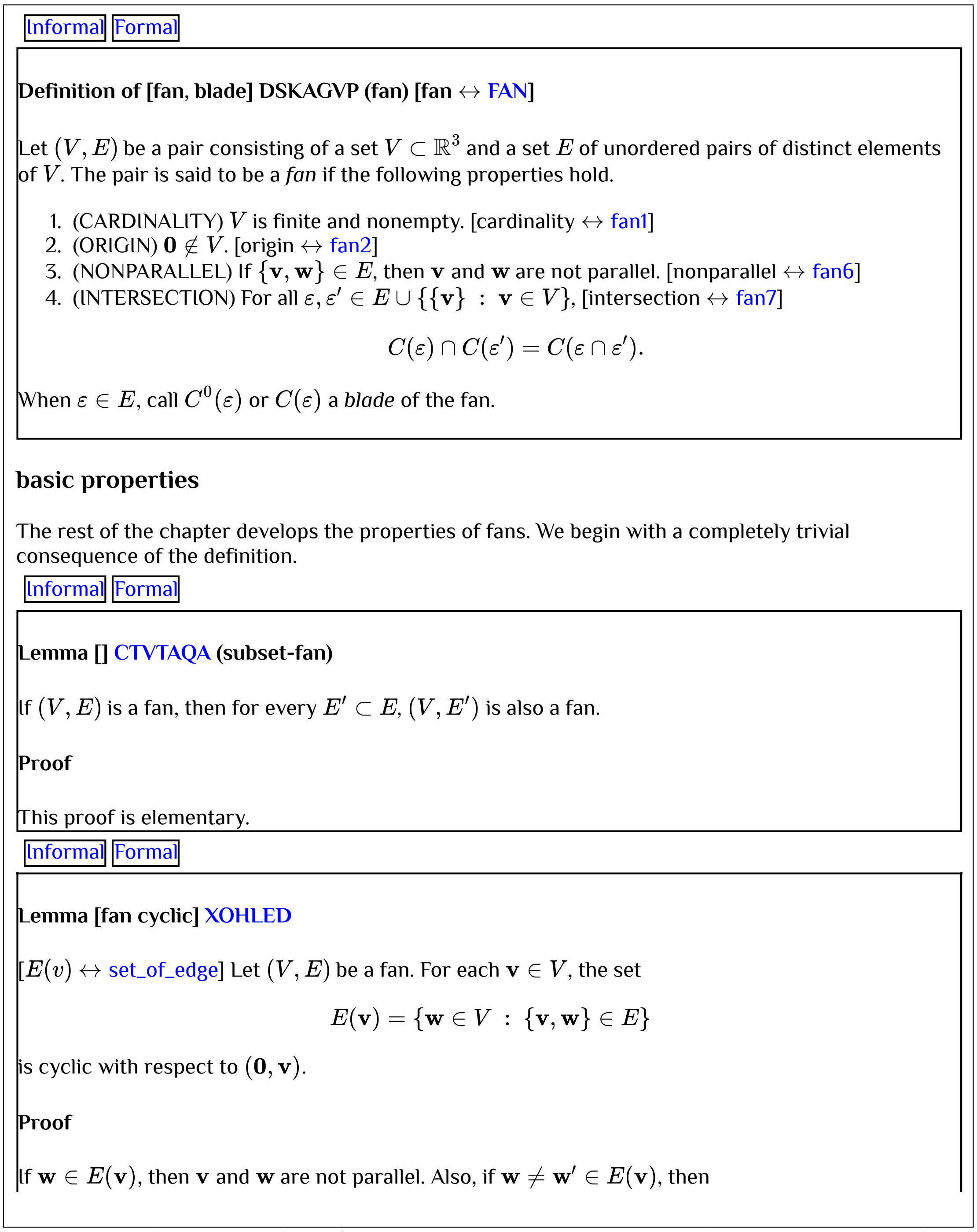
The formal text of the development, in the proof assistant syntax, is included in Agora as a set of hyperlinked HTML pages that provide *dynamic* access to the proof state, using the Proviola ([87], Chapter 3 of this thesis) technology we have previously developed: pointing at the commands in the formal text calls the proof assistant and provides the state on the page. The results of this computation are memoized for future requests: this makes it possible for future visitors to obtain these states quickly, while not taking up space unnecessarily.

The pages are hyperlinked (see Section 6.4.2) to allow a reader to explore the presented formalization. The formalization could be large and, in projects like Flyspeck, produced by a number of collaborators. The current alternatives to hyperlinking are unsatisfactory in such circumstances: it amounts to either memorization by the reader of large parts of the libraries, or mandatory access to a search facility. In HOL Light, there is no external search facility, searching is done by typing in the name of a lemma to print out its statement.

6.3 Document Structure: Frames and Scenes

The pages in Agora are generated from in-memory *documents*: (Python) objects equipped with methods for rendering and storing the internal files. To cater for multiple proof assistants and

[1] `http://mws.cs.ru.nl/agora_cicm/flyspeck/doc/fly_demo/`
[2] `http://mathjax.org`



Informal Formal

Definition of [fan, blade] DSKAGVP (fan) [fan $\leftrightarrow$ FAN]

Let (V, E) be a pair consisting of a set $V \subset \mathbb{R}^3$ and a set E of unordered pairs of distinct elements of V. The pair is said to be a *fan* if the following properties hold.

1. (CARDINALITY) V is finite and nonempty. [cardinality $\leftrightarrow$ fan1]
2. (ORIGIN) $\mathbf{0} \notin V$. [origin $\leftrightarrow$ fan2]
3. (NONPARALLEL) If $\{\mathbf{v}, \mathbf{w}\} \in E$, then $\mathbf{v}$ and $\mathbf{w}$ are not parallel. [nonparallel $\leftrightarrow$ fan6]
4. (INTERSECTION) For all $\varepsilon, \varepsilon' \in E \cup \{\{\mathbf{v}\} \ : \ \mathbf{v} \in V\}$, [intersection $\leftrightarrow$ fan7]

$$C(\varepsilon) \cap C(\varepsilon') = C(\varepsilon \cap \varepsilon').$$

When $\varepsilon \in E$, call $C^0(\varepsilon)$ or $C(\varepsilon)$ a *blade* of the fan.

basic properties

The rest of the chapter develops the properties of fans. We begin with a completely trivial consequence of the definition.

Informal Formal

Lemma [] CTVTAQA (subset-fan)

If (V, E) is a fan, then for every $E' \subset E$, (V, E') is also a fan.

Proof

This proof is elementary.

Informal Formal

Lemma [fan cyclic] XOHLED

[$E(v) \leftrightarrow$ set_of_edge] Let (V, E) be a fan. For each $\mathbf{v} \in V$, the set

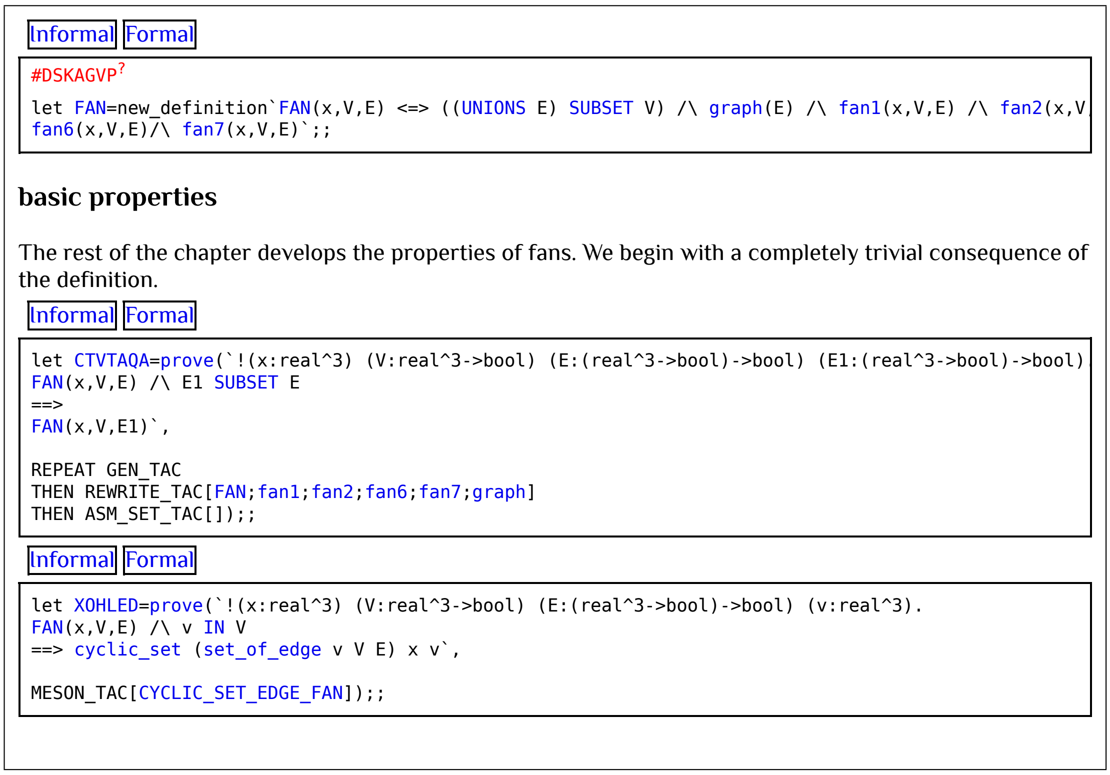
$$E(\mathbf{v}) = \{\mathbf{w} \in V \ : \ \{\mathbf{v}, \mathbf{w}\} \in E\}$$

is cyclic with respect to $(\mathbf{0}, \mathbf{v})$.

Proof

If $\mathbf{w} \in E(\mathbf{v})$, then $\mathbf{v}$ and $\mathbf{w}$ are not parallel. Also, if $\mathbf{w} \neq \mathbf{w}' \in E(\mathbf{v})$, then

Figure 6.1: Screenshot of the Agora wiki page presenting a part of the “Fan” chapter of the informal description of the Kepler conjecture formalization. For each formalized section, the user can choose between the informal presentation (shown here) and its formal counterpart (shown on the next screenshot). The complete wikified chapter is available at: `http://mws.cs.ru.nl/agora_cicm/flyspeck/doc/fly_demo/`.



Informal Formal

```
#DSKAGVP?
let FAN=new_definition`FAN(x,V,E) <=> ((UNIONS E) SUBSET V) /\ graph(E) /\ fan1(x,V,E) /\ fan2(x,V
fan6(x,V,E)/\ fan7(x,V,E)`;;
```

basic properties

The rest of the chapter develops the properties of fans. We begin with a completely trivial consequence of the definition.

Informal Formal

```
let CTVTAQA=prove(`!(x:real^3) (V:real^3->bool) (E:(real^3->bool)->bool) (E1:(real^3->bool)->bool)
FAN(x,V,E) /\ E1 SUBSET E
==>
FAN(x,V,E1)`,

REPEAT GEN_TAC
THEN REWRITE_TAC[FAN;fan1;fan2;fan6;fan7;graph]
THEN ASM_SET_TAC[]);;
```

Informal Formal

```
let XOHLED=prove(`!(x:real^3) (V:real^3->bool) (E:(real^3->bool)->bool) (v:real^3).
FAN(x,V,E) /\ v IN V
==> cyclic_set (set_of_edge v V E) x v`,

MESON_TAC[CYCLIC_SET_EDGE_FAN]);;
```

Figure 6.2: Formal links clicked at `http://mws.cs.ru.nl/agora_cicm/flyspeck/doc/fly_demo/`

document-preparation tools, such as a renderer for wiki syntax, we use the object-inheritance to instantiate documents for different systems, while providing a common interface. This interface consists of a tree-like structure of *frames*, grouped into *scenes*.

Documents in Agora are structured according to our earlier work on a system called Proviola ([83] and Chapter 3), for replaying formal proof: this tool takes a "proof script" and uses a light-weight parser to transform it into a list of separate commands. This list can then be submitted to a proof assistant, storing the responses in the process. This memoization of the proof assistant's responses is stored together with the command, into a data structure we call a *frame*. Frames can store more than just a response and a command, in particular, we assume that all frames in Agora documents store a markup element that contains the HTML markup of the frame's command.

To display a document as a page, it would be enough to display the list of frames in order, rendering the markup of each frame, and this is how the purely formal pages in Agora are rendered. However, we want our tools to be able to display not only flat lists of text, but also combine them in meaningful ways: for example by grouping a lemma with its proof, but also combining multiple lemmas into a self-contained section. For this, we introduced a *scene*: a scene is a grouping of (references to) frames and other scenes, that can combine them in any order. The system will render such a tree structure recursively, displaying the markup of each frame referenced to. The benefit of grouping files into scenes is that it becomes easier to re-mix parts of a document into a new document, such as including formal text into an informal page.

Inclusion

To allow remixing scenes from documents into new content, it is necessary to provide an interface that allows including scenes into pages. In previous work ([86] and Chapter 4 of this thesis), we introduced an interface in the form of syntax: Agora allows users to write narratives in a markup language similar to Wikipedia's, which is extended with the notion of a *reference*. This reference is similar to Isabelle's antiquotation: it is syntax for pointing to formally defined entities on the Web which carry some metadata, which can be automatically provided by a theorem prover. When rendered, the references are resolved into marked up 'islands' of formal text. The rest of the syntax is a markup language allowing mathematical notation and hyperlinks.

These islands are included in the scene structure as references to the marked up scenes. At the moment, we only allow referring to formal scenes from informal text, which is enough to render the Flyspeck text. Having an inclusion syntax fits the Agora philosophy: the documentation workflow can use the formal code, but it should not change it. Instead, writing informal documentation about a development should be similar to writing a LaTeX article, only in a different markup language. However, it is occasionally necessary to add code directly to an informal page, for example to write an illustrative example or a failed attempt; such a code block is not part of the formal development, but benefits from the markup techniques applied to the development.

In the document structure, such code blocks are just scenes, that are marked to be written in a particular language. From the rendered page, it is possible to open an editor for each scene, which requires special functionality to support writing formal proofs.

6.4 Interaction with Formal HOL Light Code

To support the formal parts of the Flyspeck project, it is necessary to add support for the language these parts are written in, HOL Light, to Agora. In particular, this means translating from HOL Light sources to the Proviola's data structure, so that previously defined operation become available for HOL Light; and adding interactive features that are only available for HOL Light. For now, the latter is restricted to proof advising.

6.4.1 Parsing and Proving

For HOL Light, adding Proviola support implies adding a parser that can transform a proof script into a list of commands, and adding a layer to communicate with the prover's read-eval-print loop (REPL). This is sufficient, but so far does not create a very illustrative Proviola display: most HOL Light proofs are *packaged* into a single REPL-invocation that introduces and discharges a theorem. Making this into a useful Proviola display is left for future work, but we will sketch how a better display can be implemented using the scene structure of a Proviola document and using the Tactician tool for HOL Light [3].

To illustrate the workings of the parser and the prover, we use the following example code, that can be obtained by unpacking a 'packaged' proof using Tactician:

```
(* Example code fragment. *)
g ‘x=x‘;;
e REFL_TAC;;
let t = (* Use top_thm to verify the proof. *)
  top_thm();;
```

Parser Because HOL Light proofs are written as syntactically correct scripts that are interpreted by the OCaml read-eval-print loop (REPL), the parser separates a proof script into the single commands that can be interpreted by this REPL. These commands are, in the Flyspeck sources, terminated by '`;;`'[3] and followed by a newline, so our parser splits a proof script into commands by looking for this terminator. Additionally, the proof can contain comments, surrounded by '`(*`' and '`*)`': we let the parser only emit a command if the terminator does not occur as part of a comment. Finally, comment blocks that are not within other commands are treated as separate commands. This last decision differs from traditional source-code parsers, which regard comments as white space, because Agora reconstructs the proof script's appearance from the frames in the movie, in order to show the complete proof script if a reader desires it.

The parser does not group the frames into a scene structure: a HOL Light proof is represented as a single scene containing all frames. For our example, the following frames are generated:

- `(* Example code fragment. *)`
- `g ‘x=x‘;;`
- `e REFL_TAC;;`

-
  ```
  let t = (* Use top_thm to verify the proof. *)
    top_thm ();;
  ```

The first comment does not occur within a command, so it is parsed as a separate command, and the second comment occurs inside a command.

[3]According to the OCaml reference manual, `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual003.html#toc4`

Command	State
`(* Example code fragment. *)`	0
`g ‘x=x‘;;`	1
`e REFL_TAC;;`	2
`let t = ...`	2

Table 6.1: Frames with state numbers

Prover HOL Light is not implemented as a stand-alone program with its own REPL. Instead, it is implemented as a collection OCaml scripts and some parsing functions. This means that the ‘prover’ instance is actually a regular OCaml REPL instance, which loads the appropriate bootstrap script. The problem of this approach is that these scripts take several minutes to load, a heavy penalty for wanting to edit a proof on the Web. To offset the load time, one can *checkpoint* the OCaml instance after it has bootstrapped HOL Light. Checkpointing software allows the state of a process to be written to disk, and restore this state from the stored image later. We use DMTCP[4] as our checkpointing software: it does not require kernel modifications, and because of that is one of the few checkpointing solutions that works on recent Linux versions.

Communication with the provers is encapsulated by a Python class: creating an instance of the class loads the checkpoint and connects to its standard input and output. The resulting object has a `send` method which writes a provided command to standard input and returns the REPL’s response. Beyond this low-level communication mechanism, the object also provides a `send_frame` method. This method takes an entire frame and sends the command stored in it. This method does not only send the text, but also records the number of tactics that the prover has executed so far, by examining the length of the current goalstack. This gives an indication of how far a list of frames is processed, and allows the prover to use HOL Light’s undo function to provide some basic state management, within a single proof.

After sending the frames generated from our example code, the frames have stack numbers as shown in Table 6.1.

When the frame with the `REFL_TAC` invocation is changed, the `send_frame` method will send the HOL Light undo function, `b ();;` as many times as is necessary to return to state 1. Afterwards, it will send the command of the changed frame.

The HOL Light glue does not send all commands equally: the Flyspeck formalization packs its proofs within an OCaml module, which causes the REPL not to give output until the module is closed. Because we want to give state information per command, the gluing code ignores the `module` and `end` commands that signal the opening and closing of modules.

Packaged Proofs To allow Proviola to record a packaged proof, it needs to break the proof down to its individual commands. To do this, we propose to use the Tactician tool [3]: this is an extension to HOL Light that records a packaged proof as it is executed, and allows the user to retrieve the actual tactics executed, which exposes the tree-like structure of such a proof: some of the tactics in the packaged proof might be applied multiple times, to different subgoals generated during the proof.

We can use the sequential tactic script generated by Tactician to generate a list of frames, that are alternatives to the commands occurring in the packaged proof. The packaged proof would render on the web interface, but when a user points at a command, the underlying

[4]`http://dmtcp.sourceforge.net`

Packaged	Tactician-unpacked
`let REAL_MUL_LINV_UNIQ = prove\n`	
`(`	
`‘!x y. (x * y = &1) ==>(inv(y) = x)‘`	`g ‘!x y. (x * y = &1) ==>(inv(y) = x)‘;;`
`,\n`	
`REPEAT GEN_TAC`	`e (REPEAT GEN_TAC);;`
` THEN\n`	
`ASM_CASES_TAC ‘y = &0‘`	`e (ASM_CASES_TAC ‘y = &0‘);;`
`THEN\n`	
`ASM_REWRITE...`	`e (ASM_REWRITE...` (on Subgoal 1)
	`e (ASM_REWRITE...` (on Subgoal 2)
`FIRST_ASSUM(...`	`e (FIRST_ASSUM...` (on Subgoal 2)
...	...

Table 6.2: Packaged proof matched to Tactician-generated tactics

Tactician-based frame would be used to display the response. To implement this, we would need to match parts of the packaged proof to the commands generated by Tactician, and store the obtained relation in the scene. For example, consider the following packaged proof:

```
let REAL_MUL_LINV_UNIQ = prove
   (‘!x y. (x * y = &1) ==> (inv(y) = x)‘,
    REPEAT GEN_TAC THEN
    ASM_CASES_TAC ‘y = &0‘ THEN
    ASM_REWRITE_TAC[REAL_MUL_RZERO; REAL_OF_NUM_EQ; ARITH_EQ] THEN
    FIRST_ASSUM(SUBST1_TAC o SYM o MATCH_MP REAL_MUL_LINV) THEN
    ASM_REWRITE_TAC[REAL_EQ_MUL_RCANCEL] THEN
    DISCH_THEN(ACCEPT_TAC o SYM));;
```

Invoking Tactician generates the following tactic-based script:

```
g ‘!x y. (x * y = &1) ==> (inv(y) = x)‘;;
e (REPEAT GEN_TAC);;
e (ASM_CASES_TAC ‘y = &0‘);;
(* *** Subgoal 1 *** *)
e (ASM_REWRITE_TAC [REAL_MUL_RZERO;REAL_OF_NUM_EQ;ARITH_EQ]);;
(* *** Subgoal 2 *** *)
e (ASM_REWRITE_TAC [REAL_MUL_RZERO;REAL_OF_NUM_EQ;ARITH_EQ]);;
e (FIRST_ASSUM (SUBST1_TAC o SYM o (MATCH_MP REAL_MUL_LINV)));;
e (ASM_REWRITE_TAC [REAL_EQ_MUL_RCANCEL]);;
e (DISCH_THEN (ACCEPT_TAC o SYM));;
```

Each individual tactic in this script, with the exception of the comments, relates to a single substring of the original proof. This means that we can match the tactics to these text positions, and obtain Table 6.4.1, which is shortened for typesetting.

When the user points at a part of the proof in the packaged proof, the goals obtained by executing the corresponding tactic in the ‘unpacked’ column are shown to the user. The table hints at several features of the relation between the scripts, that will need to be addressed in a Proviola display of a packaged proof:

- Some packed steps are matched to multiple unpacked proof steps. Because the unpacked steps are applied to different subgoals, we need to be able to show the results of these tactics on all the goals.

- Some packed steps do not match up to a single goal. These parts should either not display any dynamic behaviour, or they could display the result of executing the packed proof.

- There are tactics that are only applied to a single subgoal instead of all of the subgoals, so the matching algorithm should take care of this.

While the matching (carried out manually here) is correct, it is unclear to us if there are occasions in which the unpacked proof contains tactics that cannot be matched to a part of the packaged proof. This requires additional investigation in the workings of HOL Light and the Tactician tool.

6.4.2 Hyperlinking

To make the display of HOL Light code in Agora better suited for reader navigation, it is necessary to add markup to the code, with hyperlinks between items being the minimal required feature. To our knowledge, there is no proper hyperlinking facility for HOL-based systems so far. Such a facility should plug in to the parsing layer of the systems (as done, e.g., for Coq and Mizar), and either export the information about symbols' definitions relative to the original formal text, or directly produce a hyperlinked version of the text: this hyperlinking pass should be fast, so it can be run when a page is loaded in the browser.

For HOL Light (and Flyspeck), we so far did not try to hook into the parsing layer of the system, because this requires an intimate knowledge of the system's architecture and implementation, and only provide a heuristic hyperlinking system. Still, such a hyperlinker can be useful, because relatively few concepts are overloaded in the formalization, and most of the definitions and theorems are introduced using a regular syntax: this means that the hyperlinker can generate an index for file definitions with only a small chance of ambiguity. The hyperlinking proceeds broadly in two steps, an indexing step and a rendering step. The indexing is done by a Perl script that generates a symbol index by:

1. collecting the globally defined symbols and theorem names from the formal texts by heuristically matching the most common patterns that introduce them,[5] and
2. optionally adding and removing some symbols based on a predefined list.

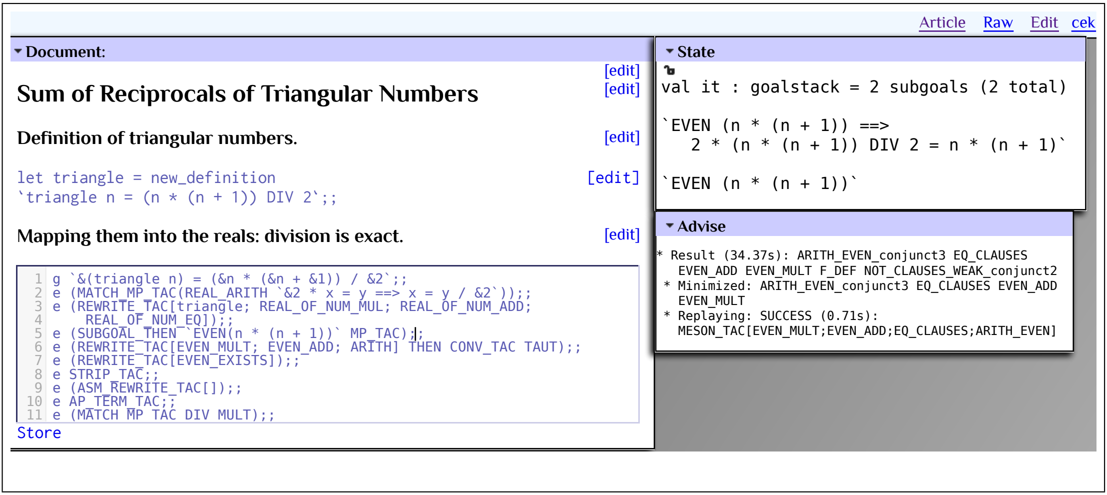
The page renderer of Agora then processes the texts again by heuristically tokenizing the text, looking up tokens and their linking in the generated index. Additionally, the page rendering also uses the index to generate metadata that can be used by the referencing mechanism [86].

The complete hyperlinking of the whole library now takes less than ten seconds, and while obviously imperfect, it seems to be already quite useful tool that allowed us to browse and study the library. The generated index of 15,780 Flyspeck entities together with their URLs can be loaded into arbitrary external application, and used for separate heuristic hyperlinking of other texts. This function is used by the script that translates the LaTeX sources of the informal text describing Flyspeck into wiki syntax (Section 6.5.2), to link the formally defined concepts to their HOL Light definitions.

6.4.3 Editing and Proof Advising

Editing We can directly use the tools that turn text into frames for building the server back-end of the web-based editor described in Chapter 5: the front end of this editor just gathers the entered text and sends it to the server, the server processes it into a list of frames and post-processes it: both by generating proof assistant (HOL Light) responses and by sending

[5]To help this, we also use the theorem names stored by the HOL Light processing in the "theorems" file, using the mechanisms from the file update_database_**.ml in the Flyspeck development.



Sum of Reciprocals of Triangular Numbers

Definition of triangular numbers.

```
let triangle = new_definition
`triangle n = (n * (n + 1)) DIV 2`;;
```

Mapping them into the reals: division is exact.

```
 1 g `&(triangle n) = (&n * (&n + &1)) / &2`;;
 2 e (MATCH_MP_TAC(REAL_ARITH `&2 * x = y ==> x = y / &2`));;
 3 e (REWRITE_TAC[triangle; REAL_OF_NUM_MUL; REAL_OF_NUM_ADD;
 4      REAL_OF_NUM_EQ]);;
 5 e (SUBGOAL_THEN `EVEN(n * (n + 1))` MP_TAC);;
 6 e (REWRITE_TAC[EVEN_MULT; EVEN_ADD; ARITH] THEN CONV_TAC TAUT);;
 7 e (REWRITE_TAC[EVEN_EXISTS]);;
 8 e STRIP_TAC;;
 9 e (ASM_REWRITE_TAC[]);;
10 e AP_TERM_TAC;;
11 e (MATCH_MP_TAC DIV_MULT);;
```

```
val it : goalstack = 2 subgoals (2 total)

`EVEN (n * (n + 1)) ==>
   2 * (n * (n + 1)) DIV 2 = n * (n + 1)`

`EVEN (n * (n + 1))`
```

```
* Result (34.37s): ARITH_EVEN_conjunct3 EQ_CLAUSES
  EVEN_ADD EVEN_MULT F_DEF NOT_CLAUSES_WEAK_conjunct2
 * Minimized: ARITH_EVEN_conjunct3 EQ_CLAUSES EVEN_ADD
  EVEN_MULT
 * Replaying: SUCCESS (0.71s):
  MESON_TAC[EVEN_MULT;EVEN_ADD;EQ_CLAUSES;ARITH_EVEN]
```

Figure 6.3: The interactive editor built in the Wiki with the proof state for the line with the cursor. The screenshot features a section of Harrison's triangular numbers formalization. In line 5 the advisor automatically finds a proof that $n(n+1)$ is even, slightly different from the one used in the edited formalization.

markup information based on the correctness of a part of the text. Because this processing is incremental, information can be returned on demand: after the text has been parsed into frames, the server can give the editor information as it is produced, using the protocols described in Chapter 5). As also described in that chapter, it remains an open question on how to properly deal with the impact of the formal text written in the editor, as this might invalidate the entire repository. An example of the editor interaction is shown in Figure 6.3. It also shows the proof advising facility described in the next paragraph.

Proof Advising In order to further facilitate the online Wiki authoring using HOL Light, we have added a post-processing step to the editor. For each goal interactively computed by the proof assistant, the editor automatically submits this goal to the AI/ATP proof advisor (HOL(y)Hammer) service [52]. The advisor uses a number of differently parametrized premise-selection methods (based on various machine-learning algorithms) to find the most relevant theorems from the Flyspeck library for a given goal, and passes them (after translation to first-order logic) to automated theorem provers (ATPs) such as Vampire [75], E [78], and Z3 [31]. If an ATP proof is found, it is minimized and reconstructed by a number of reconstruction strategies described in [53]. In parallel to such AI/ATP methods, a number of decision procedures are tried on the goal. The currently used decision procedures are able to solve boolean goals (tautologies), goals that involve naturals (arithmetic), integers, rationals, reals and complex numbers including Gröbner bases. Whenever any of the strategies finds a tactic that solves the goal, all other strategies are stopped and the result of the successful one is transmitted to the Agora users through a window. The users can immediately use the successful results in their proof.

The protocol to communicate with the advisor has been designed to be as simple as possible, in order to enable using it not only as a part of Agora but also via an experimental Emacs interface [52] and from the command line tool in the spirit of old style LCF. A request for advice consists of a single line which is a text representation of a goal to prove. To encode a goalstate as text the goal assumptions need to be separated from the goal conclusion and from each other. We use the ‘ character as such separator, since the character never appears in normal HOL Light terms as it is used to denote start and end of terms by the Camlp5 preprocessor. When a request for advice is received the server parses the goal assumptions and conclusion together, to allow matching the free variables present in more than one of them and ensure proper typing. The response is also textual and the connection is closed when no more advice for the goalstate is available. Server-side caching is used to handle repeated queries, typically produced by refactoring an existing proof script in the Wiki.

6.5 Inclusion of the Informal Flyspeck Texts

We include the informal Flyspeck texts by transforming the LaTeX sources of the text into Agora's Creole syntax. Section 6.5.1 describes the structure of the source text and the links to the formal definitions. Section 6.5.2 describes the actual translation.

6.5.1 Structure of the source text

The informal Flyspeck LaTeX text has 309 pages, but we use a smaller part of it for the experiments, namely the fifth chapter, on fans. The source file of this chapter, fan.tex has 1981 lines. There are 15 definitions (but some of them define several concepts) and 36 lemmas. The definitions have the following annotated form (developed by Hales), which already cross-links to some of the formal counterparts (formally defined theorem names like `QSRHLXB` and `MUGGQUF` and symbols like `azim_fan` and `is_Moebius_contour`). The annotations were added manually by the authors of the Flyspeck text:

```
\begin{definition}[polyhedron]\guid{QSRHLXB}
A \newterm{polyhedron} is the
intersection of a finite number of closed half-spaces in
$\ring{R}^n$.
\end{definition}
```

The lemmas are written in a similar style:

```
\begin{lemma}[Krein--Milman]\guid{MUGGQUF}
Every compact convex set $P\subset\ring{R}^n$ is the convex hull
of its set of extreme points.
\end{lemma}
```

The text contains many mappings between informal and formal concepts, e.g.:

```
\formaldef{$\op{azim}(x)$}{azim\_fan}
\formaldef{M\"obius contour}{is\_Moebius\_contour}
\formaldef{half space}{closed\_half\_space, open\_half\_space}
```

6.5.2 Transformation to Creole

There are several systems that can (to various extent) transform LaTeX texts to (X)HTML and similar formats. Examples include LaTeXML[6], PlasTeX[7], xhtmlatex[8], and TeX4ht.[9] Often they are customizable, and some of them can be equipped with custom non-HTML (e.g., wiki) renderers. For the first experiments we have however relied only on MathJaX for

[6]http://dlmf.nist.gov/LaTeXML/
[7]http://plastex.sourceforge.net/
[8]http://www.matapp.unimib.it/~ferrario/var/x.html
[9]http://tug.org/tex4ht/

rendering mathematics, and custom transformations from LaTeX to wiki syntax that allow us to easily experiment with specific functions for cross-linking and formalization without involving the bigger systems. The price for this is that the resulting wiki pages are more similar to presentations in ProofWiki and Wikipedia than to full-fledged HTML book presentations. We might switch to the larger extendable systems when it is clear what extensions are needed for our use-case.

The transformations are now implemented in about 200 lines of a Perl script that translates the Flyspeck LaTeX sources into the enhanced Creole wiki syntax used by Agora. The script is easily extendable, and it now consists mainly of about 30 regular-expression replacements and related functions taking care of the non-mathematical LaTeX syntax and macros. The mathematical text is handled by the (slightly modified) macros taken from Flyspeck (kep-macros.tex) that are prepended to any Agora Flyspeck text and used automatically by the JavaScript LaTeXrenderer MathJax. Producing and tuning the transformations took about one to two days of work, and should not be a large time investment for (formal) mathematicians interested in experimenting with Agora. The particular transformations that are now used for Flyspeck include:

- Transformations that handle wiki-specific syntax that is (intentionally or accidentally) used in LaTeX, such as comments, white space, fonts and section markup.
- Transformations that create wiki subsections for various LaTeX blocks, sections, and environments. Each definition, lemma, remark, corollary, and proof environment gets its own wiki subsection, similarly, e.g., to ProofWiki and Wikipedia.
- The transformation that add linking and cross-linking, based on the LaTeX annotations. Each LaTeX label produces a corresponding anchor, and each LaTeX reference produces a link to the anchor. Newly defined terms (introduced with the `newterm` macro) also produce anchors. Formal annotations (introduced with the `guid` and `formaldef` macros) are first looked up in the index of all formal concepts produced by hyperlinking of the formalization (Section 6.4.2), and if they are found there, such annotations are linked to the corresponding formal definition.

6.6 Conclusion and Future Work

The platform is still in development, and a number of functions can be improved and added. For example, whole-library editing, guarded by global consistency checking of the formal code that has been already verified (as done for Mizar [91]), is future work. On the other hand, the platform already allows the dual presentation of mathematical texts as both informal and formal, and the interaction between these two aspects. In particular, the platform takes both LaTeX and formal input, cross-links both of them based on simple user-defined macros and on the formal syntax, and allows one to easily browse the formal counterparts of an informal text. It is already possible to add further formal links to the informal concepts, and thus make the informal text more and more explicit. A particular interesting use made possible by the platform is thus an exhaustive collaborative formal annotation of the Flyspeck book. The platform also already includes interactive editing and verification, which allows at any point of the informal text to switch to formal mode, and to add the corresponding formal definitions, theorems, and proofs, which are immediatelly hyperlinked and equipped with detailed proof status information for every step. The editing is complemented by a relatively strong proof advice system for HOL Light. This is especially useful in a Wiki environment, where redundancies and deviations can be discovered automatically. The requests for advice can become grounds for further experiments on strengthening the advice system.

One future direction is to allow even the non-mathematical parts of the wiki pages to be written directly with (extended) LaTeX, as it is done for example in PlanetMath. This could facilitate the presentation of the projects developed in the wiki as standalone LaTeX papers. On the other hand, it is straightforward to provide a simple script that translates the wiki syntax to LaTeX, analogously to the existing script that translates from LaTeX to wiki.

Chapter 7

Conclusions

In this thesis, we have investigated the combination of web technology and proof assistant technology to support users in communicating formal mathematics. We have made a number of observations, and based on these, we have improved on the situation with tools that are part of the Agora system. We have already concluded each chapter with individual conclusions about the technologies and their futures, so we summarize in this chapter.

A proof script is not a proof Typically, a proof script for some proof assistant is distributed as *the* formal proof. For interactive theorem provers, this is not the case, as a proof script is developed in an interactive dialogue with the proof assistant, and the proof assistant output (the state) for each line of script is just as important for a reader to understand the proof. Proviola is a web display technique for formal proofs that supports this distribution model by storing input and output in a single file, and using minimal interaction to unveil output on the reader's demand.

The language of interactive theorem provers is not fit for human understanding The main goal of writing a proof in a proof assistant's language is to have that proof assistant verify the goal. To be able to, the proof assistant requires the author to write a proof down in a stricter fashion and in more detail than is required to explain a proof to a human. These extra explanatory steps are noise to a human reader. Previous approaches to this problem have focused on removing noise, either by improving the theorem provers themselves, or by creating a higher-level language that can be compiled down to a proof assistant's language.

In this thesis, we focus on supporting a language that is suitable for human understanding, but can include part of formal proofs, in any order the narrator wants to. The information used for including the intended parts of a proof is exported by knowledge the proof assistants themselves have, not making use of the verification machinery, but rather the type information that proof assistants provide. That such a hybrid model works in practice is evidenced by the case study in Chapter 6.

Web technology is mature enough to support new editing models With higher speed internet connections, the web has become an interactive playground, and software libraries, both for the browser and for the server, have grown to make use of these improvements. What we have shown in this thesis is that it is possible to connect a browser-based editor to a server-side proof assistant. The main advantage of the web editor is that it can show more contextual information than proof assistant output, and that this information can be shown as direct feedback to user edits, rather than expecting a user to 'execute' a proof step by step.

Going forwards, we see the following avenues that can improve the technology explored in this thesis, and make it more usable.

User studies The current research is based on casual observation and prototypical combinations of systems. While this is a good start for creating new ideas and seeing how well they work, it is not clear that the resulting tools are actually useful. To find this out, it is necessary to carry out *user studies*: research in the current behaviour of (potential) users of proof assistants, and their wishes for the systems.

Better designed system Agora as described in this thesis is 'PhD-ware':[1] an iteratively implemented prototype system for demonstrating the feasibility of the ideas in the thesis. This makes the system less appropriate for everyday use. To go forwards, it is advisable to redesign the system with the workflows and users in mind, and use that system for further research.

What would Agora look like in the future, if it were an ideal system? This is difficult to predict without looking in what users want, either consciously or subconsciously. What I can do here, is look at how Agora can be influenced by existing development environments for software engineering, in particular the Light Table[2] system.

Light Table is an offline integrated development environment (IDE), still in development, that provides the following ideas:

- Code can be evaluated at any point in the environment, and results are directly tied to the code that was executed to produce the result. These results are not just restricted to text, but can be anything such as web applications and games.
- When evaluating code, such as evaluating a function, the details of the evaluation are threaded through the definitions: it becomes possible to see where the given arguments of a function are used. Together with the previous idea, this is becoming known as *live programming*.
- Editing code is not constrained to editing text files. Instead, the programmer chooses parts of the code based on structure (classes, functions, methods...) and edits these parts of the code. Storing the text will eventually serialize the code into a file or more files.

These ideas can be combined in a web-based system for interactive theorem proving, the next iteration of Agora. In this system, users edit code on a per-lemma (or definition) basis, hiding lemmas and proofs that are not interesting for the proof currently being edited. To view related lemmas, the author can click on hyperlinks to these lemmas, either already on the screen or provided in a search dialog. These links would not take the user away from the editing session, but would show the lemma requested close to the editing area. Additional information, such as proof assistance output or the results of advice services, would similarly be shown in the user's work space.

This dynamic exploration and creation of a formal work is not restricted to authors of formal works, but can also be used by narrators, who write an informal text and include the

[1]This term was mentioned to the author by Alexander Serebrenik
[2]`http://www.lighttable.com`

formal text not just by writing the references from Chapter 4, but also by searching or by writing new proofs. Moreover, narrators can ‘pin’ results to highlight them for a reader or provide visualizations of certain pieces of formal code.

The reader, finally, can explore a proof much better this way, by obtaining the background knowledge needed and placing it directly in the document on screen. Moreover, a sandbox can be available at all time, to allow readers to evaluate new ideas or alternatives.

All these workflows can be supported in Agora, by taking the scenes as the elements the users manipulate: they obtain scenes through hyperlinks, modify and add scenes, and instruct the system to show certain parts of a scene, but hide others by default. To get the functionality in Agora, we need to build an interface that makes use of scenes and the tools that work on them, and presents the actions on scenes in a sensible way to the user.

The ideas described above are centered around Agora as an environment for writing and exploring formal proofs from a technical point of view. On the other hand, in recent years we have seen an increase in the use of *social* platforms for both mathematics and computer science: these systems are web pages where users can ask and answer questions about a field such as programming or group theory. This question-and-answer format leads to discussions on the best solution to a given problem. For interactive theorem proving, these discussions currently take place on mailing lists or in classrooms, but they could move to Agora: instead of having an isolated discussion on a mailing list, based on a simplified example, people could ask for help with the scenes they are writing or studying, and the resulting document can retain discussions on the formal work for future users: current mathematical discussions mainly take place near a blackboard, but future mathematical discussions can take place in Agora, centered around a living document that everyone can contribute to, and that is verified and supplemented by theorem prover computations.

Samenvatting

Wiskundige bewijzen zijn niet alleen van belang in de wiskunde, maar ook in de informatica, waar zij gebruikt worden om aan te tonen dat een programma voldoet aan bepaalde specificaties. Aangezien zulke bewijzen, zowel in de informatica als in de wiskunde, steeds vaker van een hoge complexiteit zijn, wordt het steeds moeilijker om te controleren of het bewijs *zelf* correct is. Het meest sprekende voorbeeld uit de wiskunde is het bewijs van het vermoeden van Kepler, dat stelt dat het opstapelen van sinaasappels (of kanonskogels) in een piramidevorm de minste ruimte inneemt. Het bewijs van dit vermoeden werd in 1998 geleverd door de wiskundige Thomas Hales, met behulp van een aantal computerprogramma's. Dit bewijs werd echter niet volledig geaccepteerd door het wiskundige tijdschrift *Annals of Mathematics*, omdat de recensenten de computerprogramma's niet konden controleren.

Om een hogere mate van zekerheid te krijgen in de correctheid van een bewijs, is het mogelijk deze te laten controleren door een computerprogramma, een "proof checker". Om het bewijs leesbaar te maken voor zulke programma's, is echter meer detail en een andere taal nodig dan wanneer het bewijs wordt uitgelegd aan een menselijke lezer. Deze drempels maken het moeilijk om door een computer gecontroleerde bewijzen (formele bewijzen) aan wiskundigen te laten lezen, en om samen te werken aan een bewijs. Om deze drempel te overkomen, zijn nieuwe technieken en gereedschappen nodig. In dit proefschrift wordt een aantal van zulke technieken en de bijbehorende programmatuur ontwikkeld, gebaseerd op de volgende observaties:

- Proof checkers geven (extra) inhoudelijke informatie over de bewijzen die zij verwerken. Deze informatie geeft extra inzicht aan de lezers van deze bewijzen.

- Moderne internettechnologie maakt het mogelijk informatie op een dynamische manier aan lezers te presenteren: lezers kunnen om extra informatie vragen waar zij dit nodig hebben, en bewijzen aanpassen of zelf proberen zonder dat zij andere programma's dan een webbrowser nodig hebben.

De ontwikkelde technieken komen samen in een systeem, dat via het internet te gebruiken is: Agora. Hoofdstuk 2 van dit proefschrift beschrijft het globale ontwerp van Agora en beschrijft profielen van typische gebruikers. Daarnaast schetst het een aantal methoden die gebruikt kunnen worden om deze profielen te valideren en om nieuwe mogelijkheden te ontdekken waarvoor Agora (na eventuele aanpassingen) ingezet kan worden.

Hoofdstuk 3 beschrijft de techniek waarmee Agora gecontroleerde bewijzen aan de lezer toont: de Proviola. Deze techniek geeft van een bewijs de berekening van elke stap weer *wanneer de gebruiker daar om vraagt.* Indien noodzakelijk wordt deze berekening op het moment van opvragen uitgevoerd, en opgeslagen voor later gebruik: er wordt een 'film' gemaakt van het bewijs. Een film kan zeer snel geïnspecteerd worden, zonder dat de lezer een proof checker hoeft te installeren of enige specialistische kennis nodig heeft. In dit hoofdstuk wordt ook de basis gelegd voor de rest van het proefschrift, waar de films gebruikt worden voor uitgebreidere scenario's: het maken van een beschrijvende tekst over een gecontroleerd bewijs en het schrijven van een controleerbaar bewijs via een internetpagina.

Hoofdstuk 4 toont hoe een schrijver van een formeel bewijs een beschrijving in natuurlijke taal kan maken, die verwijst naar gedefinieerde onderdelen van het formele bewijs. Deze verwijzingen worden, als de pagina aan een lezer getoond word, omgezet naar de formele tekst. De methode om deze verwijzingen te verwerken, maakt gebruik van de interne 'kennis' van de proof checker: deze weet tijdens het controleren waar een bepaalde stelling gedefinieerd is. Verder wordt gebruik gemaakt van de structuur van de films uit Hoofdstuk 3: hierin wordt het bewijs verdeeld in *scènes* die, bijvoorbeeld, een stelling bevatten. Door de informatie uit de proof checker te koppelen aan de corresponderende scène, kan Agora de stellingen op een makkelijke manier terug vinden.

In Hoofdstuk 5 ontwikkelen we een gebruikersinterface voor het schrijven van formele bewijzen. Deze interface maakt gebruik van de film als model tussen de gebruiker en de proof checker. De fim wordt automatisch gemaakt op basis van de tekst die de gebruiker intypt en verschillende gereedschappen, waaronder de proof checker, kunnen de film aanvullen met hun eigen informatie. De film wordt vervolgens gebruikt om de gebruiker te informeren over het bewijs, zonder te storen.

Hoofdstuk 6 zet Agora in voor een scenario uit de praktijk: de technieken die in dit proefschrift beschreven zijn, worden ingezet om het formele bewijs van Hales te tonen als onderdeel van zijn beschrijving in natuurlijke taal. De resulterende pagina's lijken op de oorspronkelijke beschrijving, maar bevatten de formele definities waar in het origineel alleen naar verwezen wordt. Naast het combineren van technieken uit de overige hoofdstukken, voegt dit hoofdstuk ook een integratie met een andere proof checker en een adviesdienst voor formele bewijzen toe.

Summary

Mathematical proofs are not just important in mathematics, but also in computer science, where they are used to show that a program conforms to a specification. Since these proofs, both in computer science and in mathematics, are becoming more complex, it is becoming correspondingly complex to verify if the proof *itself* is correct. One of the most telling examples from mathematics is the proof of the Kepler conjecture, which states that stacking oranges (or cannon balls) in a tetrahedral shape takes up the least space. The proof of this conjecture was given in 1998 by the mathematician Thomas Hales, who proved it aided by a number of computer programs. However, this proof was not fully accepted by the mathematical journal *Annals of Mathematics*, because the referees could not verify the programs.

To obtain a higher degree of certainty in the correctness of a proof, one can verify this with a computer program, a "proof checker". To make a proof understandable for such programs however, it requires more details and a different language than when explaining a proof to a human reader. These barriers make it difficult for outsiders to read computer-verified proofs (formal proof) and to collaborate on formal proofs. To surmount this barrier, new techniques and tools are required. This thesis develops some of these techniques and the accompanying software, based on the following observations:

- Proof checkers give (additional) information on the proofs they process. This information is insightful to the readers of these proofs.
- Modern Web technology enables readers to discover information in a dynamic manner: readers can request additional information when needed, and adapt proofs or try to write proofs themselves, without requiring any software other than a web browser.

The developed techniques are gathered in an internet-based system: Agora. Chapter 2 of this thesis describes the global design of Agora and profiles the most likely users. Additionally, it sketches some methods for validating these profiles and discover new capabilities in which Agora may serve, possibly after some modifications.

Chapter 3 describes the technique through which Agora displays verified proofs to the reader: a Proviola. This technique displays the results of each step of a proof, *when the user requests this.* When necessary, the results are computed the moment the user demands these, and stored for later use: a 'movie' is made of the full proof, that can be inspected quickly, without the reader having to install a proof checker or obtaining specialized knowledge. This chapter also lays the foundation for the rest of the thesis, in which movies are used for more involved scenarios: the writing of a narrative about a verified proof and writing a new proof, or editing a proof, directly on a web page.

Chapter 4 shows how an author of a formal proof can write a narrative in natural language, that refers to well-defined parts of a formal proof. These references are transformed into formal text when the page is displayed to a reader. The method to resolve these references uses internal knowledge of the proof checker which, during verification, knows where a certain theorem is formalized. Additionally, it uses the structure of the movies of Chapter 3, in which the proof is structured into *scenes* that for example contain a theorem. By combining the proof checker knowledge with a corresponding scene, Agora can easily retrieve the theorems.

In Chapter 5, we develop a user interface for writing formal proofs. This interface uses the movie as a model between the user and the proofo checker. The movie is automatically generated based from the text the user writes and several tools, including the proof checker, can augment with their own information. The movie is then used to inform the user about the proof without interruptions.

Chapter 6 uses Agora for a scenario drawn from the practice of formal proof: we use the techniques of this thesis to display Hales's formal proof as part of his natural language description. The resulting pages are similar to the original description, but contain formal definitions that are only textual references in the original text. Beyond combining techniques from previous chapters, this chapter also integrates a new proof checker and a proof advice service for formal proofs.

Curriculum Vitae

January 21, 1986 Born in Warnsveld, the Netherlands.

1998–2004 Gymnasium, Collegium Marianum/Valuascollege, Venlo

2004–2007 Bachelor of Science in Computer Science and Engineering, Eindhoven University for Technology. *Cum laude*

2007–2009 Master of Science in Computer Science and Engineering, Eindhoven University for Technology. *Cum laude*

- Kerckhoffs Institute specialization in Computer Security.
- Master's thesis: Classifying Attacks on Security Protocols, supervised by dr. Suzana Andova and dr. Francien Dechesne.

2009–2013 Ph.D. student, Foundations group, Institute for Computing and Information Science, Faculteit der Natuurwetenschappen, Wiskunde en Informatica, Radboud Universiteit Nijmegen.

Bibliography

[1] Resource Description Framework (RDF): Concepts and abstract syntax. Recommendation, W3C, 2004. `http://www.w3.org/TR/rdf-concepts`. 1

[2] RDFa in XHTML: Syntax and processing. Recommendation, W3C, October 2008. `http://www.w3.org/TR/rdfa-syntax`. 4.5

[3] Mark Adams and David Aspinall. Recording and refactoring HOL Light tactic proofs. In *Proceedings of the IJCAR workshop on Automated Theory Exploration*, 2012. Available at `http://homepages.inf.ed.ac.uk/smaill/atxwing/atx2012_submission_9.pdf`. 6.4.1, 6.4.1

[4] Jesse Alama, Kasper Brink, Lionel Mamane, and Josef Urban. Large formal wikis: Issues and solutions. In Davenport et al. [30], pages 133–148. 5.1

[5] Andrea Asperti, Herman Geuvers, Iris Loeb, Lionel Elie Mamane, and Claudio Sacerdoti Coen. An interactive algebra course with formalised proofs and definitions. In Kohlhase [58], pages 315–329. 4.6

[6] Andrea Asperti, Herman Geuvers, and Raja Natarajan. Social processes, program verification and all that. *Mathematical Structures in Computer Science*, 19(5):877–896, October 2009. 1

[7] Andrea Asperti, Ferruccio Guidi, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. A content based mathematical search engine: Whelp. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2004. 2.2.1

[8] Andrea Asperti and Wilmer Ricciotti. A web interface for Matita. In Jeuring et al. [48], pages 417–421. 5.1

[9] David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–42. Springer, 2000. 5.1

[10] David Aspinall, Paul Callaghan, Stefan Berghofer, Pierre Courtieu, Cristoph Raffalli, and Makarius Wenzel. Proof general. Web page, available at `http://proofgeneral.inf.ed.ac.uk/main`. 2

[11] David Aspinall, Christoph Lüth, and Burkhart Wolff. Assisted proof document authoring. In *Mathematical Knowledge Management MKM 2005, LNAI 3863*, pages 65–80. Springer, 2006. 3.5, 3.9

[12] Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors. *AISC/MKM/Calculemus*, volume 6167 of *LNCS*. Springer, 2010. 83, 91, 93

[13] Grzegorz Bancerek and Michael Kohlhase. Towards a Mizar Mathematical Library in OMDoc format. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10:23 of *Studies in Logic, Grammar and Rhetoric*, pages 265–275. University of Białystok, 2007. 4.4.1

[14] Grzegorz Bancerek and Piotr Rudnicki. A compendium of continuous lattices in MIZAR. *J. Autom. Reasoning*, 29(3–4):189–224, 2002. 4.2.1

[15] Bruno Barras and Enrico Tassi. Designing a state transaction machine for Coq. In *Coq Workshop*, 2012. 5.5.1

[16] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004. 6.1

[17] Yves Bertot. Coq in a hurry. Notes, available at `http://cel.archives-ouvertes.fr/inria-00001173`, February 2010. 3.9

[18] Alan F. Blackwell and Thomas R. G. Green. Cognitive dimensions of information artefacts: a tutorial. Tutorial, `http://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf`, 1998. 1

[19] Paul De Bra and David Smits. A fully generic approach for realizing the adaptive web. In Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors, *SOFSEM*, volume 7147 of *Lecture Notes in Computer Science*, pages 64–76. Springer, 2012. 2.2.1

[20] Sebastian Burckhardt, Manuel Fähndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's alive! continuous feedback in UI programming. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 95–104. ACM, 2013. 5.1

[21] Paul Cairns and Jeremy Gow. Literate proving: Presenting and documenting formal proofs. In Kohlhase [58], pages 159–173. 3.5, 3

[22] Adam Chlipala. Certified programming with dependent types. Draft textbook, online at `http://adam.chlipala.net/cpdt/`, 2010. 3.2, 3.5.1

[23] The Coq wiki. Browsable online at `http://coq.inria.fr/cocorico`. 4.1

[24] Alan Cooper, Robert Reimann, and David Cronin. *About Face 3 — The Essentials of Interaction Design*, chapter 5: Modeling Users: Personas and Goals, pages 75–108. In [26], 2007. 2.2

[25] Alan Cooper, Robert Reimann, and David Cronin. *About Face 3 — The Essentials of Interaction Design*, chapter 25: Errors, Alerts, and Confirmations, pages 545–547. In [26], 2007. Section: Rich visual modeless feedback. 5.1

[26] Alan Cooper, Robert Reimann, and David Cronin. *About Face 3 — The Essentials of Interaction Design.* Wiley Publishing, Inc., 2007. 24, 25

[27] Coq-Club Mailing List. The Coq-Club mailing list. Mailing List. 2.2.1, 3.3, 4.1

[28] Pierre Corbineau and Cezary Kaliszyk. Cooperative repositories for formal proofs. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *Proc. of the 6th International Conference on Mathematical Knowledge Management (MKM'07)*, volume 4573 of *LNCS*, pages 221–234. Springer Verlag, 2007. 5.1

[29] Graham Cormode, S. Muthukrishnan, and Jinyun Yun. Scienceography: the study of how science is written. *CoRR*, abs/1202.2638, 2012. 2.3

[30] James H. Davenport, William M. Farmer, Josef Urban, and Florian Rabe, editors. *Intelligent Computer Mathematics - 18th Symposium, Calculemus 2011, and 10th International Conference, MKM 2011, Bertinoro, Italy, July 18-23, 2011. Proceedings*, volume 6824 of *LNCS*. Springer, 2011. 4, 97

[31] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. 6.4.3

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison–Wesley, 1994. First edition, 20th printing. 3.4.1, 33

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*, chapter Behavioral Patterns – Observer. In [32], 1994. First edition, 20th printing. 5.5

[34] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. An interactive functional programming tutor. In Tami Lapidot, Judith Gal-Ezer, Michael E. Caspersen, and Orit Hazzan, editors, *ITiCSE*, pages 250–255. ACM, 2012. 2.3

[35] Herman Geuvers and Lionel Mamane. A Document-Oriented Coq Plugin for TeXmacs. In P. Libbrecht, editor, *MathUI workshop, MKM 2006 conference, Wokingham, UK*, `http://www.activemath.org/~paul/MathUI06/proceedings/CoqTeXMacs.html`, 2006. 3.9

[36] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *ASCM*, volume 5081 of *LNCS*, page 333. Springer, 2007. 6.1

[37] Georges Gonthier. Engineering mathematics: the odd order theorem proof. In Roberto Giacobazzi and Radhia Cousot, editors, *POPL*, pages 1–2. ACM, 2013. 1, 2.2.2, 6.1

[38] Adam Grabowski, Artur Korniłowicz, and Adam Naumowicz. Mizar in a nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010. 6.1

[39] Rob Gravelle. *Comet Programming: Using Ajax to Simulate Server Push*, 2009. Example tutorial taken from the web. 5.4.2

[40] Thomas C. Hales. *Dense Sphere Packings - a blueprint for formal proofs.* Cambridge University Press, Sep 2012. 1, 6, 6.2

[41] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010. 6.1

[42] John Harrison. HOL Light: An overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *LNCS*, pages 60–66, Munich, Germany, 2009. Springer-Verlag. 6.1

[43] Marijn Haverbeeke. Implementing a syntax-highlighting Javascript editor—in Javascript /* a brutal odyssey to the dark side of the DOM tree */. Blog post, May 2007. 5.3.2

[44] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space.* Morgan & Claypool, 2011. 10

[45] Carsten Heinz and Brooks Moses. The listings package. Technical report, CTAN, 2007. `http://www.ctan.org/tex-archive/macros/latex/contrib/listings`. 2

[46] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant – a tutorial. Web page, available at `http://coq.inria.fr/getting-started`., February 2007. 3.9

[47] The Isabelle mailing list. `cl-isabelle-users@lists.cam.ac.uk`. 2.2.1, 4.1

[48] Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors. *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science.* Springer, 2012. 8, 86

[49] Cezary Kaliszyk. Web interfaces for proof assistants. In Serge Autexier and Christoph Benzmüller, editors, *Proceedings of the FLoC Workshop on User Interfaces for Theorem Provers (UITP'06), Seattle*, volume 174[2] of *Electronic Notes in Theoretical Computer Science*, pages 49–61, 2007. 3.3, 5.1, 5.3.2

[50] Cezary Kaliszyk. *Correctness and Availability. Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* PhD thesis, Radboud University Nijmegen, 2009. 1.1, 3.2

[51] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *CoRR*, abs/1211.7012, 2012. 6.1

[52] Cezary Kaliszyk and Josef Urban. Automated reasoning service for HOL Light, 2013. Accepted to CICM 2013. 6.4.3

[53] Cezary Kaliszyk and Josef Urban. PRocH: Proof reconstruction for HOL Light, 2013. Accepted for CADE 2013, `http://mws.cs.ru.nl/~urban/proofs.pdf`. 6.4.3

[54] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *Communications of the ACM*, 53(6):107–115, jun 2010. 1

[55] Gerwin Klein, Tobias Nipkow, and Larry Paulson (editors). The Archive of Formal Proofs. Online Journal, 2004–. `http://afp.sourceforge.net`. 6.1

[56] Donald E. Knuth. *Literate Programming.* The University of Chicago Press, 1992. 3.5

[57] Andrea Kohlhase and Michael Kohlhase. Maintaining islands of consistency via versioned links. In *Proceedings of the 29th ACM international conference on Design of communication*, SIGDOC '11, pages 167–174, New York, NY, USA, 2011. ACM. 2.4.2

[58] Michael Kohlhase, editor. *MKM'05*, number 3863 in LNAI. Springer Verlag, 2006. 5, 21, 90

[59] Michael Kohlhase. *OMDoc - An Open Markup Format for Mathematical Documents [version 1.2]*, volume 4180 of *Lecture Notes in Computer Science*. Springer, 2006. 4.4.1, 6.1.1

[60] Christoph Lange. OMDoc ontology. `http://kwarc.info/projects/docOnto/omdoc.html`, 2011. 4.4.1

[61] Christoph Lange. Ontologies and languages for representing mathematical knowledge on the semantic web. *Semantic Web Journal*, 2012. In press. 4.4.1

[62] Christoph Lange and Josef Urban, editors. *Proceedings of the ITP 2011 Workshop on Mathematical Wikis (MathWikis)*, number 767 in CEUR-WS, 2011. 4, 87

[63] Bo Leuf and Ward Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley Professional, 2001. 1.1

[64] Andres Löh. lhs2TeX. Web page, available at `http://people.cs.uu.nl/andres/lhs2tex/`, December 2009. 3.9

[65] Christoph Lüth and Martin Ring. A web interface for Isabelle: The next generation. In Jacques Carette, editor, *Conferences on Intelligent Computer Mathematics CICM 2013*, volume 7961 of *Lecture Notes in Artificial Intelligence*, pages 326–329. Springer, 2013. 5.1

[66] Lionel Mamane. *Interactive Mathematical Documents: Creation and Presentation*. PhD thesis, Radboud Universiteit Nijmegen, 2013. 3.2

[67] The Mizar mailing list. `mizar-forum@mizar.uwb.edu.pl`. 2.2.1, 4.1

[68] The Mizar wiki. Browsable online at `http://wiki.mizar.org`. 4.1

[69] Bengt Nordström. Towards a theory of document structure. In Yves Bertot, Gerard Huet, Jean-Jacques Levy, and Gordon Plotkin, editors, *From Semantics to Computer Science: Essays in Honor of Gilles Kahn*, chapter 12, pages 265–279. Cambridge University Press, 2008. Available at `http://www.cs.chalmers.se/~bengt`. 3.9

[70] Oracle Corporation. Javadoc tool homepage. Web page, available at `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`, 2010. 3.5

[71] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007. 6.1.1

[72] Benjamin C. Pierce, Chris Casinghino, and Michael Greenberg. Software foundations. Course notes, online at `http://www.cis.upenn.edu/~bcpierce/sf/`, 2010. 2.3, 3.1, 3.2, 3.5.1, 3.9, 5.2, 5.6

[73] Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford Univ. Press, 1998. 1

[74] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013. 2.2

[75] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002. 6.4.3

[76] Christoph Sauer, Chuck Smith, and Tomas Benz. Wikicreole: a common wiki markup. In *WikiSym '07*, WikiSym '07, pages 131–142, New York, NY, USA, 2007. ACM. 2.4.1, 2.4.2, 4.3, 4.6

[77] Simon Schenk and Paul Gearon. SPARQL 1.1 Update. W3C working draft, World Wide Web Consortium (W3C). 2.2.1

[78] Stephan Schulz. E - A Brainiac Theorem Prover. *AI Commun.*, 15(2-3):111–126, 2002. 6.4.3

[79] Heinrich Stamerjohanns, Michael Kohlhase, Deyan Ginev, Catalin David, and Bruce R. Miller. Transforming large collections of scientific publications to xml. *Mathematics in Computer Science*, 3(3):299–307, 2010. 2.4.1

[80] William A. Stein et al. Sage mathematics software, 2009. 6.1.1

[81] Carst Tankink. Proof in context — web editing with rich, modeless contextual feedback. In Cezary Kaliszyk and Christoph Lüth, editors, *Proceedings of the 10th International Workshop On User Interfaces for Theorem Provers*, 2012. 1.4, 5, 6.1

[82] Carst Tankink, Herman Geuvers, and James McKinna. Narrating formal proof (work in progress). *Electr. Notes Theor. Comput. Sci.*, 285:71–83, 2012. 1.4, 3.1

[83] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk. Proviola: A tool for proof re-animation. In Autexier et al. [12], pages 440–454. 1.4, 3.1, 5.3, 6.3

[84] Carst Tankink, Cezary Kaliszyk, Josef Urban, and Herman Geuvers. Communicating Formal Proofs: The Case of Flyspeck. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 451–456. Springer, 2013. 1.4, 6

[85] Carst Tankink, Cezary Kaliszyk, Josef Urban, and Herman Geuvers. Formal Mathematics on Display: A Wiki for Flyspeck. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *MKM/Calculemus/DML*, volume 7961 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2013. 1.4, 6

[86] Carst Tankink, Christoph Lange, and Josef Urban. Point-and-write - documenting formal mathematics by reference. In Jeuring et al. [48], pages 169–185. 1.4, 4, 5.1, 6.2, 6.3, 6.4.2

[87] Carst Tankink and James McKinna. Dynamic proof pages. In Lange and Urban [62]. 1.4, 6.2.2

[88] The Coq Development Team. The Coq proof assistant. Web page, obtained from `http://coq.inria.fr` on October 5, 2009. 3.2, 5.1

[89] The homotopy type theory blog. `http://homotopytypetheory.org/`. 4.1

[90] Josef Urban. XML-izing Mizar: making semantic processing and presentation of MML easy. In Kohlhase [58], pages 346–360. 4.5

[91] Josef Urban, Jesse Alama, Piotr Rudnicki, and Herman Geuvers. A wiki for Mizar: Motivation, considerations, and initial prototype. In Autexier et al. [12], pages 455–469. 1.3, 4.5, 6.6

[92] Josef Urban, Piotr Rudnicki, and Geoff Sutcliffe. ATP and presentation service for Mizar formalizations. *J. Autom. Reasoning*, 2012. 5.6

[93] Josef Urban and Geoff Sutcliffe. Automated reasoning and presentation support for formalizing mathematics in Mizar. In Autexier et al. [12], pages 132–146. 4.1, 4.5

[94] Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors. *ITP 2011*, volume 6898 of *LNCS*. Springer, 2011. 3

[95] Bill Venners. Exploring with wiki — a conversation with Ward Cunningham, part 1. interview, October 2003. Response to the second question. 5.1

[96] Bret Victor. Inventing on principle. Invited talk at the Canadian University Software Engineering Conference (CUSEC), January 2012. 5.1

[97] Makarius Wenzel. Isabelle as document-oriented proof assistant. In Davenport et al. [30], pages 244–259. 4.5, 5.1

[98] Makarius Wenzel. Read-eval-print in parallel and asynchronous proof-checking. In *User Interfaces for Theorem Provers 2012*, 2012. 5.4.1, 5.5.1

[99] Makarius Wenzel. PIDE as front-end technology for Coq. *CoRR*, abs/1304.6626, 2013. 5.1

[100] Markus M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002. 4.3.1, 4.3.2

[101] Freek Wiedijk. The De Bruijn factor. In *Proceedings of the 2000 TPHOLs conference*, 2000. 3.2

[102] Freek Wiedijk, editor. *The Seventeen Provers of the World, Foreword by Dana S. Scott*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006. 1

[103] Freek Wiedijk. Pollack-inconsistency. *Electr. Notes Theor. Comput. Sci.*, 285:85–100, 2012. 1

Titles in the IPA Dissertation Series since 2007

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination.* Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain.* Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems.* Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2013-02

E. Zambon. *Abstract Graph Transformation – Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2013-03

B. Lijnse. *TOP to the Rescue – Task-Oriented Programming for Incident Response Applications.* Faculty of Science, Mathematics and Computer Science, RU. 2013-04

G.T. de Koning Gans. *Outsmarting Smart Cards.* Faculty of Science, Mathematics and Computer Science, RU. 2013-05

M.S. Greiler. *Test Suite Comprehension for Modular and Dynamic Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2013-06

L.E. Mamane. *Interactive mathematical documents: creation and presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2013-07

M.M.H.P. van den Heuvel. *Composition and synchronization of real-time components upon one processor.* Faculty of Mathematics and Computer Science, TU/e. 2013-08

J. Businge. *Co-evolution of the Eclipse Framework and its Third-party Plug-ins.* Faculty of Mathematics and Computer Science, TU/e. 2013-09