Narrating Formal Proof (Work in Progress)

Carst Tankink Herman Geuvers James McKinna

Institute for Computing and Information Science
Radboud University
Nijmegen, The Netherlands

Abstract

Building on existing work to proxy interaction with proof assistants, we have considered the problem of how to augment this data structure to support **commentary** on formal proof development. In this setting, we have studied extracting commentary from an online text by Pierce et al. [11].

Keywords: Coursebooks, Proof Assistants, Proof Communication

1 Introduction

Much research in user interfaces for Proof Assistants (PAs) has gone into facilitating the *authoring* of proof documents. However, the communication of proof scripts to outsiders, such as mathematicians or students, has in our view not received the attention it deserves.

In this paper we consider a method and tools for enriching a proof document for communication to such third parties. The enhancement of the document consists of adding a marked-up narrative to the document and including the PA responses for dynamic display.

As a running example, we consider the writing of coursebooks used in teaching *with* a PA, especially the course notes of Pierce et al. on Software Foundations [11]. From these course notes, we can extract the markup using Coqdoc, and insert the PA responses using the concept of *movies*, introduced by us in a recent paper [12]. In this setting, we briefly sketch how to add editable exercise environments to proof documents.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

2 Background

2.1 Scenario

In this paper, we consider a scenario of communication: an author of a (formal) proof document wants to communicate this to a reviewer, who might not have prior experience in a PA and is definitely not an expert in the system. This restriction on reviewer expertise means that for him to interpret a proof document, one or more of the following should hold:

- the proof document 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 document (a proof script in this case) can be loaded in a PA, so the reviewer can evaluate the effects of each tactic on the general proof state; or
- the proof language in which the document is written mimics closely the vernacular of informal mathematics.

To bring things into focus, we consider specific instances of author, reviewer and PA here:

Author The author in this paper will be an author writing a coursebook for use in a computer science curriculum. The book does not necessarily have to teach the use of a PA, but can present a formal model of (a slice of) computer science that is verified by the PA.

Reviewer The reviewer then becomes the prime consumer of a coursebook: a student taking the course. We assume the student has no prior experience with the PA used to write the coursebook.

PA For concrete examples and tools, we choose Coq [13] as our PA: this choice is motivated by local expertise in the Coq system and tools, and the existence of at least two coursebooks written as a Coq script. These books are "Software Foundations" by Pierce et al. [11] and "Certified Programming with Dependent Types" by Chlipala [4]. Despite this choice, we believe that the techniques illustrated here are also applicable to other PAs, especially tactic-based ones.

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

- 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 tactics 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:

(i) 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.

(ii) The script itself, loaded in an interface to the PA such as CoqIDE (part of the Coq distribution) or ProofGeneral [1]. 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 reviewer in understanding a proof document: describing a proof using a high-level narrative and reviewing the proof script dynamically, by loading it in a PA 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 PA: to our knowledge, no interface to a PA actually renders the documentation of a proof document in a nice way, and the rendering does not incorporate the PA output based on reader focus. Additionally, installing and configuring a PA requires effort of the reviewer, an effort that we have lightened by integrating script and output in a single document, a proof **movie**.

2.2 Movies

A proof movie is a self-contained recording of the interaction between a user and a PA (for further details, please see our recent manuscript [12]). The PA responses can then later be retrieved from the movie without recomputation. The movie can be used to communicate the contents of a proof script without the reader needing to install and configure a PA, nor recompute the proof state.

The movie data structure is a list of frames. In its most basic form, a frame ties together the command sent to the PA and the response of the PA to this command. We have implemented the movie as an XML file, with frame, command and response as node types.

Watching a movie

Watching a movie is done by viewing an HTML rendering of its contents. The script responsible for transforming the XML into HTML is dubbed **Moviola**, after an editing tool for physical film. The page presents the command part of each frame, creating a view that is similar to the proof script sent to the PA. When the reviewer places his cursor on a command, the corresponding response is obtained from the movie and shown to the reviewer.

Watching a movie requires no sophisticated tools: all that is needed is the movie, the XSL script transforming the XML into HTML and a web browser. Additionally, instead of publishing the XML together with an XSL file, a stand-alone XSL processor can also be used to generate an HTML file. This HTML file can then be loaded into the browser.

Constructing a movie

Construction of a movie can be done either as a post-processing step of a proof script, or interactively.

The post-processing of a script is done by splitting up the script into individual commands and sending these commands to the PA. The responses are subsequently recorded into the frame.

Interactively constructing a movie is done by giving an author a view of the unfinished movie. In this view, it is possible to insert new commands and edit old ones, while the PA can insert responses to these commands, which are shown to the author, if requested. In this way, the author and the PA cooperate in constructing a movie, consisting of a proof script and the responses to the tactics in the script.

The main benefit of the movie is that it cuts out the PAs computation when a reader wants to see the response to a specific command, at the cost of not having a certified answer. The resulting movies are just plain text, however, not enhanced with the pretty rendering provided by tools such as Coqdoc.

2.3 Adding narrative: Coqdoc and others

To create pretty-printed documentation for proof scripts, there are broadly two categories: either one can use specific syntax to write documentation inside the proof script (typically as comments), or one can write a higher-level document from which both script and documentation can be extracted. The latter approach is also known as *literate proving* and allows the author to write both documentation and proof in tandem.

Coqdoc is the Coq version of the first approach. Distributed together with the Coq PA, 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 extracts special comments from the document. 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.

As an example of the second approach, Aspinall, Lüth and Wolff [2] 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 tools and the proof author. The tools can extract relevant information from the text, and also insert information back into the document, through the concept of *backflow*. Example tools are a PA, that takes tactics and can insert proof state, or LaTeX-related tools, that create PDF out of the narrative. To insert PA data inside the narrative, an author can use a command to insert a placeholder for the proof state, which is later replaced by the PA's actual 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. In the next section, we investigate how we might improve the Coqdoc-produced pages by adding a movie-reel to it.

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 both tools, although the PG kit approach might be adapted to support the scenario.

2.4 Course notes

We have decided to focus on coursebooks for education using a PA, and as a specific case study, we will look at the course notes by Pierce et al. for a course on Software Foundations taught at the University of Pennsylvania [11]. 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:

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

(ii) At the deepest level of the document tree, the subsections have paragraphs as leaves. These leaves can be either slices of proof script or paragraphs in the narrative.

(iii) The proof script forms a special structure outside the structure of the text, that of a sequential set of commands interpretable by a PA.

Chlipala has also written a coursebook, one on dependently typed programming [4], but we do not focus on it here, beyond the observation that he includes PA output as part of the narrative, reinforcing our belief that it is desirable to perform the interleaving of movie and rendering.

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.

3 Enhancing movies with commentary

A movie is a sequential series of frames, which do not contain the pretty rendering. This rendering, provided by Coqdoc, can easily be integrated in the movies. To do so, we created a tool that takes the commands from the frames and feeds these commands to Coqdoc. Coqdoc outputs an HTML tree for the command, that contains more information about the intention of the command. In particular the tree can have nodes of the following types:

- 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 nodes produced by Coqdoc are added to the frame as additional data, that can be used for several purposes.

3.1 Rendering enhanced movies

Instead of displaying the plain text of a movie, we can display the rendered text as created by Coqdoc instead. This display 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 through the web, and we have provided a web page displaying the course notes dynamically. The page can be found at `http://mws.cs.ru.nl/moviola/movies/coqdoc`. Despite the obvious limitations of including static screenshots here in order to illustrate a dynamic feature, Figure 1 displays the effect of placing the cursor on a tactic.

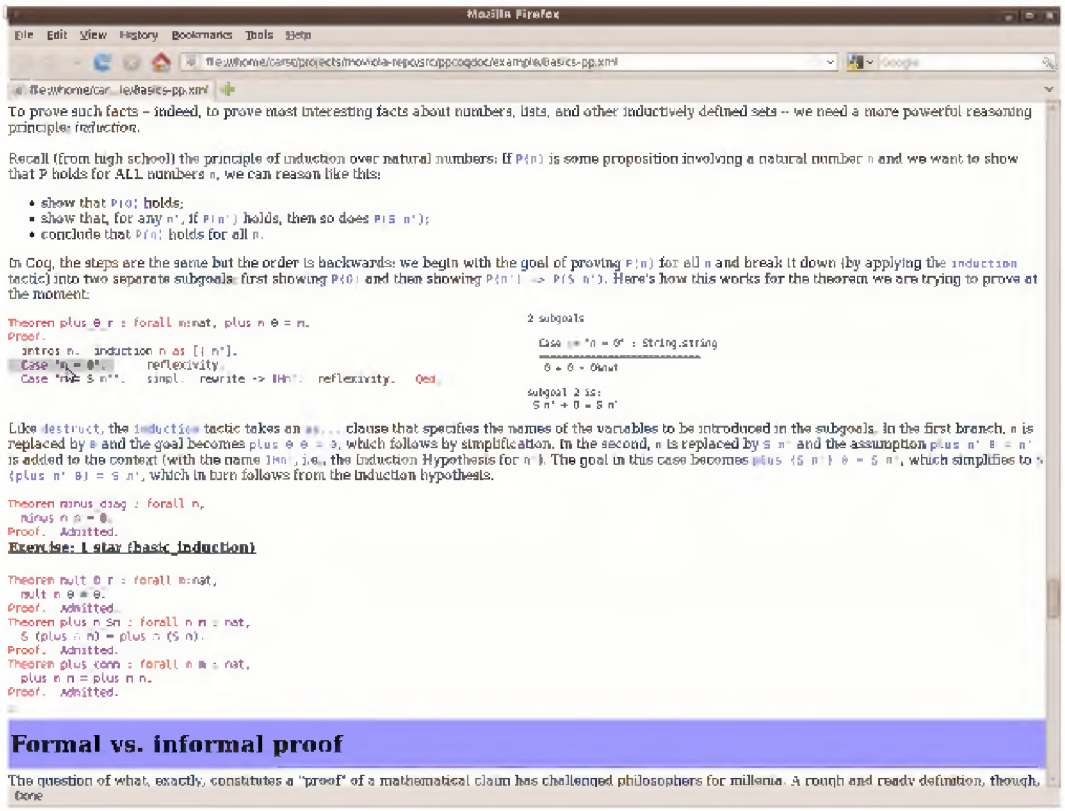

Figure 1. A screenshot of the movie

3.2 Scenes

These rendered pages do not have the structure associated with the narrative of a coursebook built in: it still is just a sequence of frames, only now rendered prettily. For further analysis and better structuring, we can group a set of frames into a *scene*.

A scene in a movie mirrors the section of an article. As such, it can contain the following data:

Text Text is just that: the narrative of the document. It can be rich text, including HTML markup and Unicode characters, but has no interactivity or structuring.

Scenes To further structure the movie, a scene can contain sub-scenes, just as sections can contain subsections for further structuring.

Code frames Beyond the normal text, a scene can contain frames. Each frame contains a single command from the proof script and the corresponding response from the PA. The display of the response is dynamic: only the commands are shown, and when a reviewer places the cursor on a command, the response is shown.

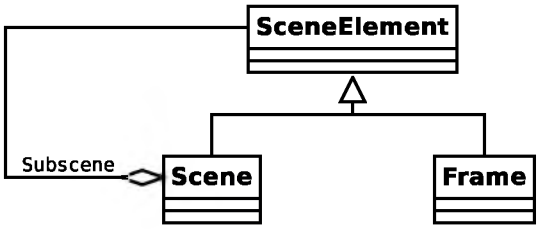

Figure 2. Class diagram of a scene

The architecture of a scene is an instantiation of the Composite pattern, its class diagram is displayed in Figure 2.

Because explanation within the narrative can refer to future or previous sections and recapitulate, or abstract from, previous fragments, it seems desirable that scenes can refer to other scenes freely, beyond the rigid structure noted above.

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 both paragraphs and section headers.

The headers can be used to group the paragraphs and frames following it, up to the next header. If this header is of a 'lower' level (for instance: a subsection header following a section header), the frames following the subheader is a sub-scene of the scene being built, and if it is of the same or 'higher' level, we go up to this higher level, finishing all the scenes of a lower level.

With the sketched recursive algorithm, we can simply group the frames of the movie into a nested structure mimicking the structure of the document. Additionally, it seems useful to group subsequent sequences of command frames into their own scene. More specifically, grouping the proof of a lemma or theorem into a scene seems the most logical, but this requires looking at the text of the commands itself, instead of the data on the structure of the HTML tree.

4 Adding Commentary to a Proof

For rendering, a scene is a minimal addition, making the output to web pages a bit easier, but the real advantage for having scenes is in post-processing data: a scene forms a logical entity within the narrative, that might be enriched with specific metadata or be edited further. In particular, writing commentary *after* the script has been made can be supported by first grouping a set of frames into a scene, and then describing this scene as a whole.

To write such a **commentary track** for a movie, an author needs the following:

- A movie created from a proof script.
- An interface through which she can write the commentary track, and tie it to the frames.

We are still experimenting with the interface for writing the commentary track, but based on the data structure and an initial prototype, we observe that the interface should provide for the following activities:

- Writing the actual text.
- Grouping code frames and text into scenes.
- Interleaving text and code to obtain 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 does not introduce new HCI problems.

The first design decision to be made is how to allow an author to group text into frames. As the resulting document structure is a tree, a tree editor could be used for adding scenes to the document, or to select 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.

How to interleave the text and code is not yet clear to us. To make the scenes as flexible as possible, we decided that the relation between frames and scenes should be many-to-many: code and narrative are equally important, and it is not unlikely that the narrative refers to a previous definition or skips forward to a proof or lemma. It proves difficult to design an interface that allows creating this many-to-many relation without forcing the author to a specific workflow.

The state-of-the-art in programming environments might be useful to borrow ideas from, but approaches like Javadoc [10] are normally used to document programs on the level of classes, methods and interfaces. In a proof setting, this would translate to documenting a lemma instead of describing chunks of commands.

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. 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

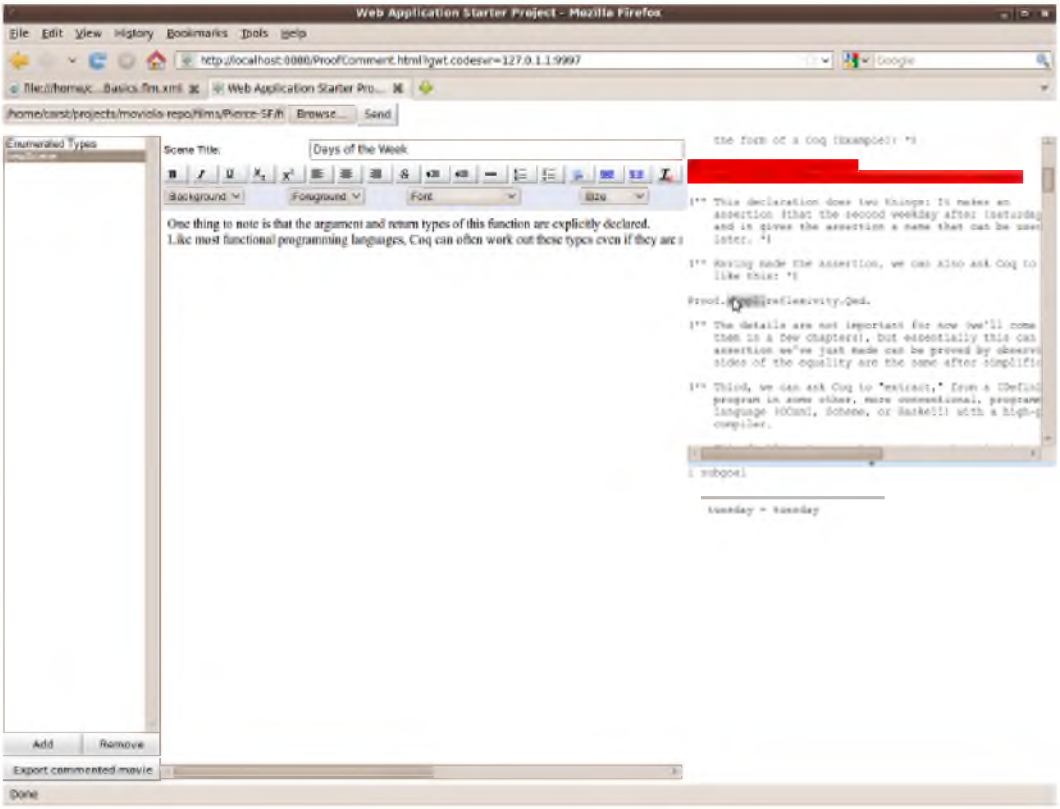

Figure 3. A screenshot of the commentary tool

her to toggle scene inclusion by a click on the desired scenes. A screenshot is shown in Figure 3.

This interface forces the user in a rather restricted workflow: she would first need to add a scene, then alternate between typing and choosing the code to be included. Furthermore, it does not allow her to interleave the code within the narrative. For now, improving the user interface for writing commentary is left as an open issue.

5 Interactive movie elements

Although we have added dynamic content to Coqdoc documents, this does not make a proof document really *interactive*: the content of the movie does not change in response to a reader's actions, only its display does. We now consider how we can add interactive scenes to a movie, without having to give the reviewer full access to the proof script or requiring him to load a PA.

In our chosen context of course notes, the main way of providing an interactive version these notes is by providing exercises: a given set of theorems and definitions that still have holes in them that the reader can fill in. An actual

PA supports doing exercises unsupervised by checking a proof once it is done, and by providing the state after each command, which helps in progressing through the proof.

These holes are intended to be filled in by the student, leading to a fully checked proof document. On the other hand, the explanation in a text for students should not have to be edited by those students. To allow the distinction between exercises and text, we would like to have **editable scenes** in the movie. In this section, we propose an as of yet unimplemented design for such scenes.

5.1 Writing Editable scenes

An editable scene is a scene that can 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 PA processing the commands the reader types in an exercise scene.

Note that 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.

5.2 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 for editing these scenes. This would include marking the scene as editable, for example by providing an edit button next to the scene, and by including a PA-backed editor for filling out the exercise.

We have not attempted to design such an editor, but we would prefer it to be very light-weight: the workflow of reading the document should not be disrupted too much by doing the exercise. Because of this, we do not want the student to switch to another page for filling out an exercise. This means we would like the following use case to be fulfilled by the editor:

(i) The student clicks the 'edit button' .

(ii) The movie's server brings a PA into the state necessary for doing the exercise

(iii) The editor is shown to the student, including the PA's state (context and

goals) for the exercise.

(iv) In the editor, the student types commands, which update the PA's state.

(v) If the student solves the exercise, it is stored, if he abandons it, the exercise gets abandoned.

To implement the communication with a PA, we would use the ProofWeb system [7], developed at Nijmegen. ProofWeb is a client-server architecture for doing formal proof over the web. At the server side, PAs are installed, that can be communicated with through a JavaScript client. Instead of the provided UI, we could build our own lightweight editor, and connect that to the ProofWeb server.

The main open problem is handling the PA state: before the editor is shown, quite some computation is necessary to bring the PA 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 PA as a background process, stopping at the first exercise. This is a naive, but probably easily implemented solution, that does not account for exercises being skipped or abandoned.
- To handle a student skipping an exercise, we could tacitly insert an `Admitted` command for every exercise. Once a student has solved it, we then remove the admission. This would work for Coq, but we do not know if all PAs support an `Admitted`-like construct. Apart that, the computation to get to the focused exercise might become too slow, as the student might start with the last exercise, requiring the entire chapter to be sent to the PA in order to start the exercise.
- We could be smarter about the inter-proof dependencies: most PAs 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 PA needs to be more permissive about the proof structure, or our tool support could build a sequence of commands from the path to the selected leaf.
- 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, but to our knowledge, no PA supports this behaviour, and getting this behaviour with external tools seems difficult.

6 Related work

Leading up to this paper, we have created a dynamic version of the Software Foundations course notes [11]. We have applied our techniques to create handouts for a PA 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. [6] and the tutorial by Bertot [3].

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 [2]. Additionally, Mamane and Geuvers have experimented with a document-oriented Coq plugin for TeXmacs [5], and lhs2TeX [8] allows writing literal proof documents, from which both Coq code and LaTeX documentation can be extracted. 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 [9] using dependent type theory to enforce syntactic wellformedness of books and articles, 'live' documents, programs, and formal proofs in a unified way. Especially his notion of typed placeholders could be used to represent exercises in a online coursebook.

7 Conclusions

We have shown how we can make on-line coursebooks using a PA more dynamic: by adding the PA'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 dynamic documents could be created from a proof script when the script itself cannot be modified, 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.

References

[1] Aspinall, D., P. Callaghan, S. Berghofer, P. Courtieu, C. Raffalli and M. Wenzel, *Proof general*, Web page, available at http://proofgeneral.inf.ed.ac.uk/main.

[2] Aspinall, D., C. Lüth and B. Wolff, *Assisted proof document authoring*, in: *Mathematical Knowledge Management MKM 2005, LNAI 3863* (2006), pp. 65–80.

[3] Bertot, Y., *Coq in a hurry*, Notes, available at http://cel.archives-ouvertes.fr/inria-00001173 (2010).

[4] Chlipala, A., *Certified programming with dependent types*, Draft textbook, online at http://adam.chlipala.net/cpdt/ (2010).

[5] Geuvers, H. and L. Mamane, *A document-oriented Coq plugin for TeXmacs*, in: P. Libbrecht, editor, *MathUI workshop, MKM 2006 conference, Workingham, UK*, http://www.activemath.org/~paul/MathUI06/, 2006.

[6] Huet, G., G. Kahn and C. Paulin-Mohring, *The coq proof assistant – a tutorial*, Web page, available at http://coq.inria.fr/getting-started. (2007).

[7] Kaliszyk, C., *Web interfaces for proof assistants*, in: S. Autexier and C. Benzmüller, editors, *Proceedings of the FLoC Workshop on User Interfaces for Theorem Provers (UITP'06), Seattle*, Electronic Notes in Theoretical Computer Science **174**[**2**], 2007, pp. 49–61.
URL http://www4.in.tum.de/~kaliszyk/docs/cek_p2.pdf

[8] Löh, A., *lhs2TeX*, Web page, available at http://people.cs.uu.nl/andres/lhs2tex/ (2009).

[9] Nordström, B., *Towards a theory of document structure*, in: Y. Bertot, G. Huet, J.-J. Levy and G. Plotkin, editors, *From Semantics to Computer Science: Essays in Honor of Gilles Kahn*, Cambridge University Press, 2008 pp. 265–279, available at http://www.cs.chalmers.se/~bengt.

[10] Oracle Corporation, *Javadoc tool homepage*, Web page, available at http://java.sun.com/j2se/javadoc/ (2010).

[11] Pierce, B. C., C. Casinghino and M. Greenberg, *Software foundations*, Course notes, online at http://www.cis.upenn.edu/~bcpierce/sf/ (2010).

[12] Tankink, C., H. Geuvers, J. McKinna and F. Wiedijk, *A Moviola for proof re-animation*, Submitted to the 9th International Conference on Mathematical Knowledge Management (MKM 2010) (2010), available through http://cs.ru.nl/~carst/files/moviola.pdf.

[13] The Coq Development Team, *The Coq proof assistant*, Web page, obtained from http://coq.inria.fr on October 5, 2009.