Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 285 (2012) 71–83

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Narrating Formal Proof (Work in Progress)

Carst Tankink Herman Geuvers James McKinna

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

Abstract

Building on existing work in proxying interaction with proof assistants, we have previously developed a *proof movie*. We have now considered the problem of how to augment this movie 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 (formal) proof documents. However, the *communication* of proof texts to outsiders, such as mathematicians or students, has in our view not received the attention it deserves.

In previous work [12], we presented the **movie** data structure that contains both a finished proof script and the PA responses to the commands in the script; each response is linked to the command in a **frame**. A movie is rendered as a webpage, that dynamically displays, *without further interaction with the PA*, the response to a given command when the reader hovers over it.

In this paper, we discuss our ongoing work in enriching the movie data structure with explanatory narrative, and the methods and tools we are developing to support such 'commentary' (which we think of as akin to multiple possible 'soundtracks' to a real movie).

A first approximation is to emulate existing practice using the Coqdoc (available as part of the Coq distribution [13]) or JavaDoc [10] style, whereby specially marked comments are extracted from an existing proof script to form the narrative. These paragraphs are then used to display a marked up (or 'narrated') movie, that retains

doi:10.1016/j.entcs.2012.06.007

the dynamic quality of original movies, combined with the narrative of a literate proof script. We have experimented with this first approach, using the course notes of Pierce et al. on Software Foundations [11] to achieve an almost identical set of pages, but augmented with the dynamic visualisation of the proof process.

A narrated proof movie obtained in this manner is, however, still *linear*, or rather, retains the same ordering as those of the original frames of the underlying movie: the narrative is extracted from a linear script, and is not capable of recalling previous frames or alluding to future ones.

To improve on this situation, we consider in this paper the overlay of an additional structure, that of a **scene**, that contains both a narrative and a list of frames: within a scene, there is no *a priori* order imposed on either the narrative or the (references to the) commands: the list of frames need neither be contiguous nor order-respecting with respect to the original movie. We currently are experimenting with a prototype implementation of scene overlay.

We have examined these ideas in the specific setting of writing coursebooks for use in teaching *with* a PA, as for example the course notes of Pierce et al. previously mentioned. In the setting of course notes, we briefly sketch how we might wish to add editable exercise environments to proof documents.

2 Background

We briefly recap our 'proof movie' set-up. For further details, please see our recent paper [12]. We considered various use cases for a PA, identifying (and distinguishing) the producer, or **author** role, and that of the consumer, or **reader**. The author produces a finished **script** of commands for the PA, which is then processed by a **camera** to produce a **movie**, a data structure implemented as an XML document that contains a linear list of **frame**s. In its most basic form, a frame ties together the command sent to the PA and the response of the PA to this command. The rendering engine for movies is called a **proviola**: as the reader navigates back-and-forth over the text of the proof script, the PA response to a given command is displayed dynamically. For *re-checking* of the script, it is a simple erasure process to recover the actual commands from a movie for resubmission to the PA.

2.1 Scenario

In this paper, we consider the following scenario: the author wishes to communicate a script to a reader, who might not have prior experience with a PA 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

PA, 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 paper we further consider specific instances of author, reader and PA:

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 PA, but can present a formal model of (a slice of) computer science that is verified by the PA.

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 PA used to write the coursebook.

PA For concrete examples and tools, we focus on the Coq system and its associated toolset [13]: 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. [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 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:

(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 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 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 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. The tmEgg tool [5] is an attempt to do this, but requires the document to be written and viewed using the TEXmacs editor. Additionally, installing and configuring a PA 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.

Our original design for movies just contained plain text, not enriched with any rendering. To enhance the presentation of movies, we consider replacing the actual contents of the script with a pretty-printed version. In the case of Coq, we do not write our own pretty printer, but use the Coqdoc tool.

2.2 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 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.

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 the author and by tools. Example tools are a PA, taking tactics and inserting proof state, or LATEX-related tools, creating 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, to be replaced later by actual PA 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 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.

2.3 Course notes

We have decided to focus on coursebooks for education using a PA, 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 [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 commands to the PA 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.

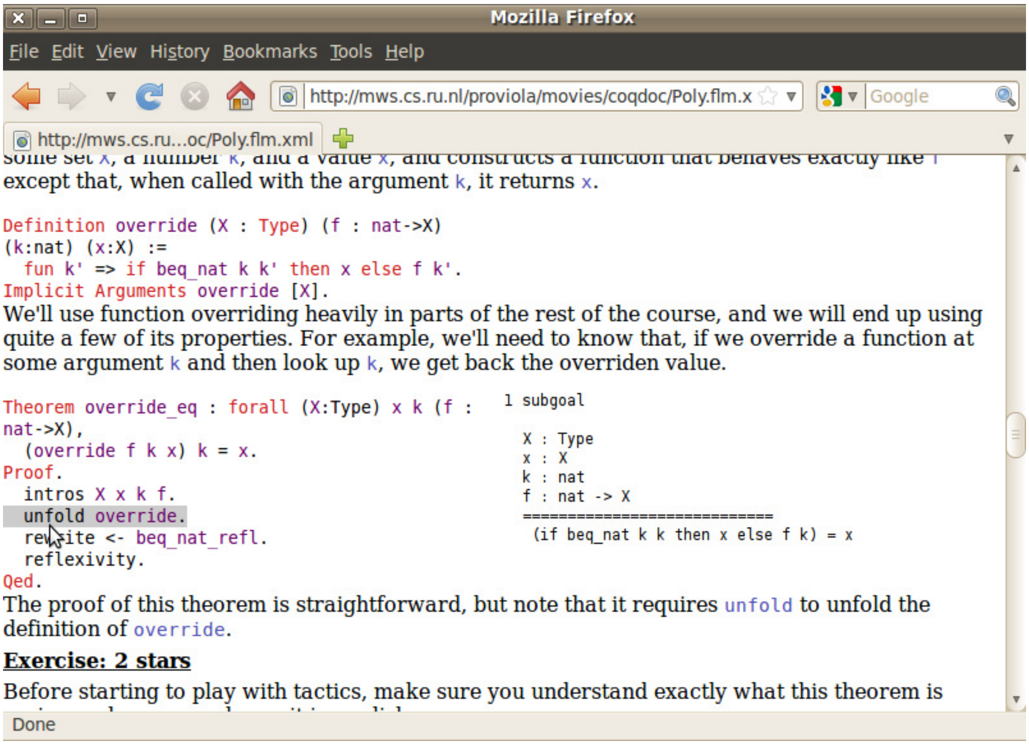
Chlipala has also written a coursebook, one on dependently typed programming [4], but we do not focus on it here, beyond the observation that he includes PA output as part of the narrative, reinforcing our belief that it is desirable to perform the interleaving of movie and rendering.

3 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:

- Documentation nodes, further structured in:
  - · section headers, for different section levels,
  - · narrative paragraphs, containing the text of the commentary.


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

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 1. A screenshot of the movie

- 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, especially for displaying the paragraphs nicely.

3.1 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/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: 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.

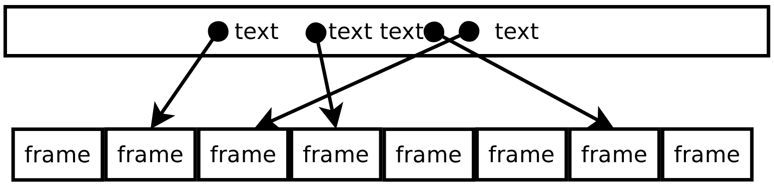


Figure 2. A scene forms a comb structure on the movie

3.2 Scenes

Narrative extracted from a script has one 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.

As an alternative to the rigid structure, we plan to introduce **scenes** to the movie. 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:

- the narrative text, marked up where appropriate; and
- references to frames. These references are contained within the text, and cause the corresponding frames to be displayed when the movie is 'watched'.

By this design, the scene 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 2.

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

(i) select the frames that the scene describes (creating the teeth of the comb);

(ii) write the text, including references to the selected scenes (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.

We have not yet implemented this design, but do not expect any issues to arise. As a first prototype, our tools group the code frames which follow each other into their own scene. As future enhancement, we want to group the proof of a lemma or theorem into a single scene, but this requires looking at the contents of command nodes, instead of just the structure of the HTML tree.

4 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 are still experimenting 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.

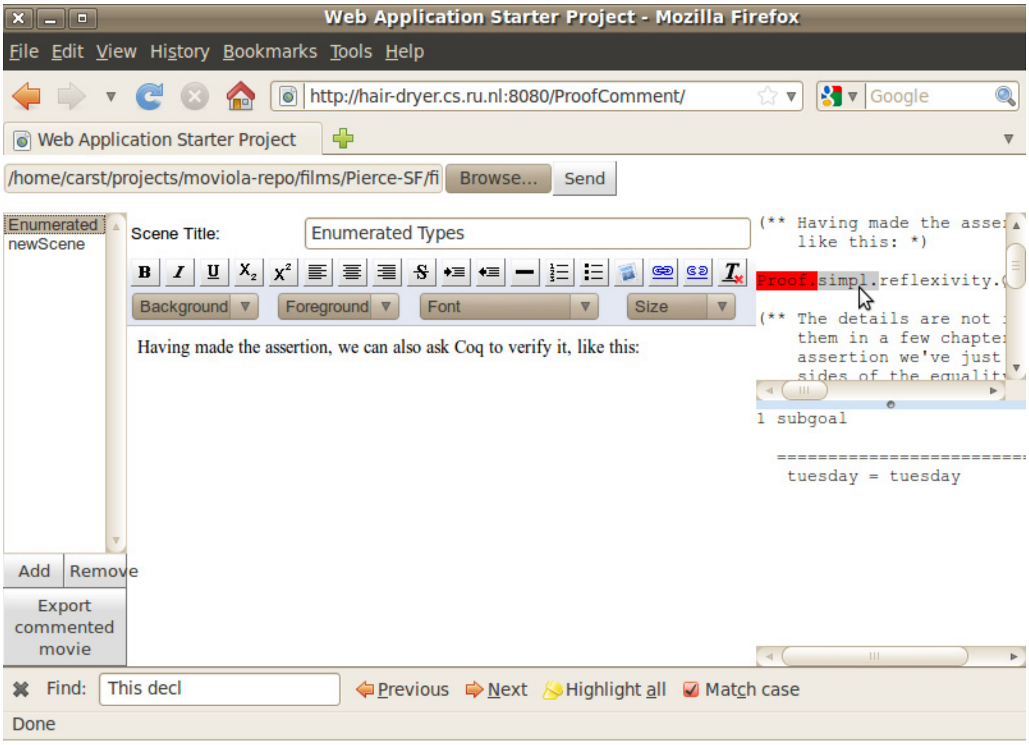
On the other hand, inferring the movie's structure when the author inserts a header might provide a faster editing workflow, as adding a new scene does not require her to switch to a different menu or editor.

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; 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



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

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

Enumerated
newScene

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. A screenshot of the commentary tool

workflow: she would first need to add a scene, then alternate between typing and choosing code to be included.

The prototype does not yet provide the author with a means of referring to the selected frames. Exactly how to support this remains ongoing work.

5 Outlook: Making movies interactive

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 PA, 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 PA 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 have an edit box and a connection to a (remote) PA. This PA can interpret what the reader types in the edit box, and insert feedback, in the form of proof state, into the movie.

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.

5.1 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 PA 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.

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 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 PA-backed editor for filling out the exercise.

We have not yet designed such an editor, but we would prefer it to be very lightweight: 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 efficiently: 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 not all PAs support an `Admitted`-like construct, so this solution cannot be easily adapted to other PAs that we might want to adapt our technology to. Apart from 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 external tools could submit only the commands that are necessary to get to the selected leaf to a PA.
- 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. The HOL family of PAs supports this behaviour, but we have not yet focused on these systems.

Once we have the machinery for checking proof text in place, we can further investigate how to support (unsupervised) e-learning using PAs: 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.

6 Related work

Leading up to this paper, we 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], and the tmEgg experimental document-oriented Coq plugin for TEXmacs by Mamane and Geuvers [5]. Hinze and Löh's lhs2TEX [8] allows writing literal proof documents, from which both Coq code and LATEX documentation can be extracted. 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. 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. In particular, 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 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.

Acknowledgement

We are grateful to Benjamin Pierce and colleagues for making their extensive course notes available, and for their encouragement of our experiments. We would like to thank the anonymous referees, whose questions and suggestions helped improve this paper. This research is partially funded by the NWO project "MathWiki: a Web-based Collaborative Authoring Environment for Formal Proofs", the NWO BRICKS/FOCUS project "ARPA: Advancing the Real use of Proof Assistants" and the NWO cluster DIAMANT.

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, Wokingham, UK*, `http://www.activemath.org/~paul/MathUI06/proceedings/CoqTeXMacs.html`, 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, *Proviola: a tool for proof re-animation*, Accepted for 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.