Interactive Proof Presentations with *Cobra*

Martin Ring
DFKI
Bremen, Germany
martin.ring@dfki.de

Christoph Lüth
DFKI and Universität Bremen
Bremen, Germany
christoph.lueth@dfki.de

We present *Cobra*, a modern code and proof presentation framework, leveraging cutting-edge presentation technology together with a state of the art interactive theorem prover to present formalized mathematics as active documents. *Cobra* provides both an easy way to present proofs and a novel approach to auditorium interaction. The presentation is checked live by the theorem prover, and moreover allows live changes both by the presenter as well as the audience.

1 Introduction

Presenting formalized mathematical proofs is a challenge unto itself. Formalized proofs, by their very nature, tend to be lengthy, as every possible side condition and assumption has to be tackled, and often the amount of proof text devoted to the main argument is small in relation to those necessary but tedious side issues. Further, the actual proof usually contains technical information such as setting up the syntactic machinery and automatic proof assistance of the prover, which is not necessary for the comprehension of the mathematical argument, but necessary if the audience wants to rerun the proof in the actual prover. Thus, when presenting formalized proofs to students, fellow researchers, or other audiences, we only present a *view* on the actual proof text. This view only contains excerpts of the original proof, and it is left to the punctiliousness of the presenter to not introduce errors. The correlation between the presentation and the actual proof text is left to the critical audience to verify. Further, when made with traditional presentation aids (such as PowerPoint, LaTeX, or writing on plain old blackboards) these presentations are not interactive — we cannot easily change them during the presentation, *e.g.* to demonstrate why a particular approach will not prove the desired goal. However, a proof made with an *interactive* theorem prover is an *active document* and should be treated, and presented, as such. The stop-gap measure often used up to now has been to switch back and forth between the presentation and the actual running prover, but the resulting change of focus makes it hard to follow the proof. Another fix is to animate the presentation manually by means of PDF overlays or PowerPoint animations, but this is inflexible, error-prone and cumbersome.

Fortunately, the advances of modern web technologies have opened up new ways of presentation, which allow us to treat a proof as an active document rather than an inanimate piece of PDF. Tools such as reveal.js implement presentations as active documents, and thus it seems logical to leverage this technology for interactive provers. In this paper, we present *Cobra*, an integrated presentation environment for interactive proofs and code. *Cobra* allows to declaratively define interactive slides containing Isabelle theories (or snippets from a theory), LaTeX-style formulae and program code. The intriguing aspect about these slides is that they can not only be presented with annotated semantic information provided by the underlying prover (or compiler in the case of code), but also the content can be altered (or completed), resulting in updated semantic annotations. This way, the presenter can develop the proof in front of, and even in interaction with, the audience, instead of following a strict predetermined path.

To appear in EPTCS.

The paper is structured as follows: Section 2 describes how to work with *Cobra* from a users perspective. Section 3 briefly sketches the architecture of the project. We conclude in Section 4 by describing proposed use cases and providing an outlook onto future work.

2 Presenting with *Cobra*

Creating slides is one of the most time consuming tasks in the preparation of a lecture or a talk. Thus it should involve no additional overhead, to allow the speaker to focus on content rather then technical details. *Cobra* aims to be even simpler than presenting with the LaTeX beamer class but a lot more powerful. It comes as a self-contained command-line tool for all major operating systems, and supports Isabelle/HOL, Scala and Haskell as well as LaTeX-style formulae out of the box, and furthermore can be extended to suit individual needs.

Creating an empty *Cobra* presentation is as easy as running `cobra new <name>` on the command line. This creates a directory `<name>` containing two files: `cobra.conf` and `slides.html`. The former contains meta information and environment configurations, while the latter contains the actual content of the presentation. The user may add other files into the directory, which will then be served by *Cobra*. This allows to include graphics, videos, custom style sheets and other arbitrary files, which can then be included in the `slides.html` file. During presentation, *Cobra* will run an embedded web server, and serve the slides as HTML pages (see Section 2.5).

2.1 Configuration

The `cobra.conf` file is a HOCON (*Human-Optimised Config Object Notation*) configuration file. There is a `reference.conf` included in the distribution which contains all available settings together with default values and short descriptions. There is no need to change the `cobra.conf` for most presentations. However, among others, the customisations shown in Table 1 are available. Other available settings include configuration of the underlying presentation framework reveal.js, the math engine *MathJax* as well as Isabelle. All settings provide reasonable default values ("convention over configuration").

`title`	display title of the presentation
`language`	main language of the presentation
`theme.slides`	main style sheet that should be used to render the slides.
`theme.code`	main style sheet that should be used to render code snippets.
`binding.interface`	network interface to bind the server on (default `localhost`)
`binding.port`	port under which the server will be available (default `8080`)

Table 1: Some of the settings available through `cobra.conf`

2.2 Adding slides

Cobra is based on the reveal.js framework [2]. However, in *Cobra* the `slides.html` file is not structured like a common web page: it does not contain a top level `html` tag or any header tags. All boilerplate is automatically generated by *Cobra*. Slides are represented by top-level `section` tags and behave the same as reveal.js slides with the exception of `code` tags, which are described in section 2.4.

```
<code class="hidden"
      src="src/Seq.thy"></code>

<section>
  <h2>Lemma</h2>
  <code class="state-fragments"
        src="#lemma-reverse"></code>
  <p class="fragment">
    Note: $$reverse_conc$$ is required here
  </p>
</section>
```

(a) `slides.html`

```
theory Seq
import Main
begin
...

(* begin #lemma-reverse *)
lemma reverse_reverse: "reverse (reverse xs) = xs"
  by (induct xs) (simp_all add: reverse_conc)
(* end #lemma-reverse *)

end
```

(b) `Seq.thy`

Figure 1: Simple example with just one slide and external Isabelle source

reveal.js has a two dimensional slide layout: It is possible to nest `section` tags by one level, which results in vertically arranged slides. This can be used to group slides together. The content of the `section` tags can be arbitrary HTML, allowing for rich presentations. However, typically a small subset will suffice. Particularly interesting HTML Tags are headers (`h1`, `h2`, ...) for the title of a slide; `img` tags to include vector or raster graphics; unordered and ordered lists (`ul`, `ol`) for bullet lists. reveal.js supports so called fragments, which allow to unhide or emphasise certain parts of the slide. Fragments are added through class tags. Listing 1a contains a small example with just one slide.

2.3 Integrating LaTeX-style formulae

By surrounding text with double dollar signs (i.e. `$$...$$`) it is interpreted as LaTeX and rendered by *MathJax*, a JavaScript library that can render MathML and a large subset of TeX/ LaTeXand is compatible with all modern browsers [4]. The MathJax configuration can be altered through the `cobra.conf` file.

2.4 Including Proofs and Code

As mentioned above `code` tags are treated specially by *Cobra*. By default, every `code` tag induces an editable code snippet. By adding certain classes to the tag, the behaviour can be altered: The language mode can be defined by adding either `scala`, `haskell` or `isabelle` as a class tag. When such a tag is added, the a rich semantic assistance engine is automatically initialised to provide information about the snippet, that will be visualised during the presentation.

The simplest way to include code is to add it as content of a `code` element. However, it is often desirable to include only certain snippets of a larger code example. For these situations, *Cobra* allows to include hidden `code` elements and then reference snippets. Snippets are marked by special comments within the target language, which is more robust than referencing lines. Snippets may overlap. When dealing with large code examples it also possible to include just a reference to an external source file. In this case, the language does not have to be specified as it can be derived from the file ending.

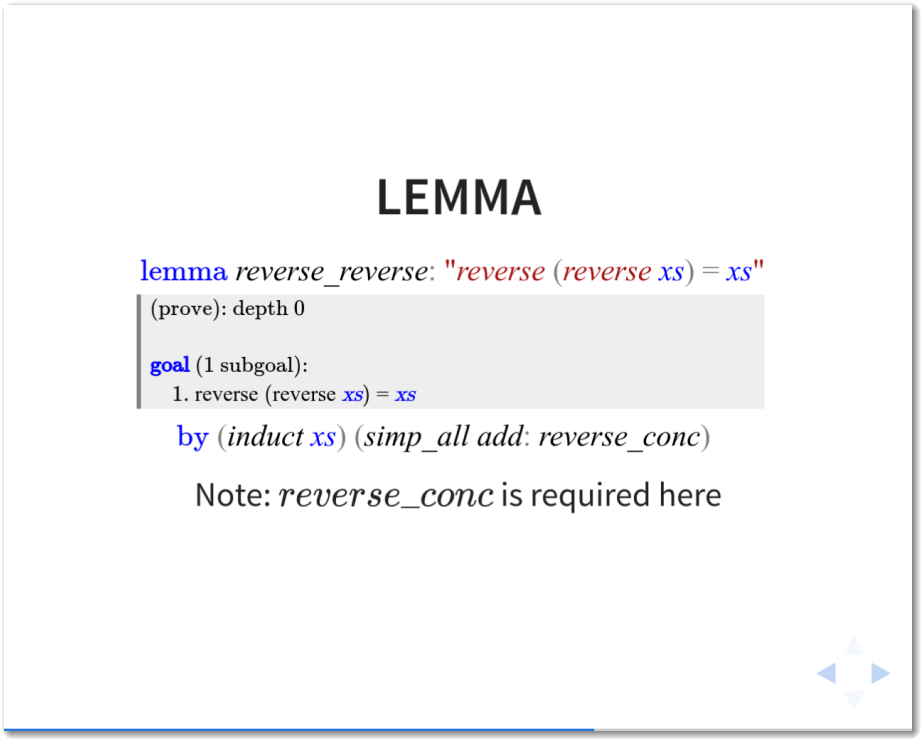

Figure 2: The rendered result of Listing 1a

2.5 Interactive Presentations

To start a presentation, the user calls `cobra <dir>` in the command line to start a presentation server within the specified directory. After a short while *Cobra* will be initialised and display the bound address under which the presentation is available. It can now be opened with any modern web browser. When open, the user will see the first slide. It is possible to navigate through the slides with the arrow keys or a presenter just like with any other presentation tool. reveal.js provides controls similar to those of PowerPoint; an overview of the available key bindings can be displayed by pressing `"?"`.

When a slide includes a code snippet, the viewer will find it augmented with annotated semantic information. The syntax highlighting will reflect the semantic meaning of tokens, just like in Isabelle/jEdit or Clide. For Isabelle theories, the proof states can be iterated (just like other fragments) if the class `state-fragments` is added. It is also possible to mark certain fractions of the code as fragments which will behave like any other reveal.js fragment but also affect the semantic annotations. Errors are prominently rendered as red sketchy circles around the affected part of the snippet, which is useful when presenting a situation where an error is of importance. It is also possible to view additional information about entities by hovering with the mouse (or a finger when using a touch enabled device). The displayed tool-tips are displayed in a way that is suitable for presentations, and contain similar information to the tool tips in Isabelle/jEdit. Since manually hovering can be cumbersome it is also possible to define *hover-fragments*, which automate this by allowing to step through predefined hover locations.

Furthermore, it is possible to alter the content of snippets just like in a text editor[1]. When snippets

[1]Note that, for security reasons, the Safari web browser does not allow keystrokes in full screen mode, which unfortunately makes the browser unsuitable for presenting interactive proofs

are changed, the content is synchronised across all connected devices as well as the underlying assistant, which then annotates the code with updated semantic annotations, similar to the non-human collaborators in Clide. It is also possible, that someone from the audience alters the code with his own device. The changes will then be synchronised to everybody else and become visible in the main presentation. Of course, this is only possible if the speaker allows access to the presentation server. Snippets are synchronised continuously; *e.g.* if there is one running code example that is referenced across various slides, changes made in one slide might affect the others. Overlapping snippets will always display consistent information. If this is not desired, the snippets have to be separated into different origins.

2.6 Publishing and Distribution

There are three intended ways of publishing slides. All three have unique advantages and disadvantages. Thus, it is desirable to provide at least two different ways to access slides for the audience.
Central Presentation Server: The presentation can be provided on a central presentation server which then also runs the Isabelle process. Viewers can access the presentation and play around with proofs through their web browser. This approach requires a lot of resources on the presentation server when many viewers access different presentations at the same time.
Local Presentation Servers: It is possible to distribute a *Cobra* presentation as an archive file. This will require the viewers to install the *Cobra* command line tool as described above, and install Isabelle locally.
PDF Export: Having active slides is not always desirable, especially when it comes to just viewing and printing. In this case it is possible to export PDF. This can be done the same way as in reveal.js; *Cobra* then takes care of the rendering of snippets for print.

3 *Cobra* internals

The PIDE framework which was developed together with the Isabelle/jEdit integration [10] has already been successfully integrated into the collaborative web environment *Clide* in previous work [7]. We were able to reuse significant portions of the code to implement *Cobra*. However, due to the limitations at the time, the *Clide* server is implemented in Scala, a strongly typed and reasonably well specified language, while the client was implemented in CoffeeScript, a thin wrapper language compiling directly to JavaScript. Both are dynamically typed languages, with an object system based on prototyping, and lacking a module system. While this makes these languages suited for "quick and dirty" solutions, complex projects like *Clide* or *Cobra* become hard to maintain and reason about. Due to the recent leaps in the complexity of average web applications, several independent approaches, like TypeScript [3], arose to remedy these shortcomings. A particularly interesting solution in our context is ScalaJS [5], a JavaScript back-end for the Scala compiler, which compiles Scala code to JavaScript which can run in a web browser; it has officially been labelled "ready for production" by the creators of Scala in 2015.

Thanks to ScalaJS the *Cobra* code base does not include a single line of JavaScript. We do depend on JavaScript libraries like CodeMirror (the editor component) or reveal.js (the underlying presentation framework), but were able to create Scala facade types from existing TypeScript types. Especially the basis of *Clide*, the collaboration algorithm, is now identical on the JVM and all connected browsers.

System Architecture. *Cobra* is designed as a client-server architecture. Unlike *Clide* which is based on the Play! framework, the *Cobra* server is built around akka-http [8], a minimal HTTP library based

on the Akka implementation of reactive streams [1]. This results in a reduced size and better speed of the application. Upon connection, web clients load all static assets from the *Cobra* server through plain HTTP, these include the compiled JavaScript of the client. After that, a WebSocket connection is established which handles all further communication. The protocol is a very fast and size-efficient binary protocol based on the boopickle library, which is itself derived from the Scala Pickling project [6]. This allows to pass Scala defined algebraic data types around between client and server, both of which are implemented in Scala. Since the PIDE framework is also implemented in Scala, the integration is straightforward. The same holds for the Scala compiler. Haskell is integrated by calling ghc-mod as an external command. To synchronise document states across clients and assistants, the universal collaboration approach from Clide is used (as described in [7]).

4 Conclusion

We have presented *Cobra*, a framework which allows source code and interactive proof scripts to be presented as active documents, with which the viewer can interact. We believe, that our tool is ready for use in production. Version 1.0 will be released prior to the workshop and made available on the GitHub project page (`https://github.com/flatmap/cobra`).

We envisage the following usages scenarios for *Cobra*: Firstly, classroom teaching where a prepared proof is presented to a group of students. Here, the advantages of *Cobra* are that the teacher can select those parts of the proof to be presented, so the presentation remains compact and convenient to follow, avoiding cognitive overload with lots of unnecessary details. The teacher can further build up the proof gradually, like one a blackboard, but with the safety net of the Isabelle proof checker in the background. Secondly, teaching in small groups where the teacher develops a proof together with a group of students. Here, the proof can be developed collaboratively, with everybody contributing while the presentation serves as the focus of attention for all participants. Thirdly, self-study when readers (students, fellow researchers, or reviewers) can interact with the presentation, exploring the effects of changes. Finally, research talks at a workshop such as UITP, where researchers can present their work with greater confidence, and can pick up questions from the audience by demonstrating effects of changes as they might be suggested from the floor. In all of these scenarios, *Cobra* brings added value over the current state of the art, where interactive proofs are presented as passive documents.

In *future work*, we plan to include other proof assistants. The canonical way to connect other proof assistants is via the PIDE framework. As a PIDE integration exists [9], it should be feasible to integrate Coq. In addition, it would be beneficial to reduce the overhead further by introducing a simplified syntax to describe slides, possibly based on Markdown. Further, we plan to explore if more inter-dependencies between the snippets and the presentation itself could be allowed. This would allow for even richer presentations, where the structure of a presentation depends on the contained proofs or generated graphics visualise dynamic aspects of a proof.

References

[1] *Reactive Streams*. `http://www.reactive-streams.org/`. Accessed: 2016-05-16.

[2] *reveal.js website*. `http://lab.hakim.se/reveal-js`. Accessed: 2016-05-16.

[3] Gavin Bierman, Martin Abadi & Mads Torgersen (2014): *Understanding typescript*. In: *ECOOP 2014–Object-Oriented Programming*, Springer, pp. 257–281.

[4] Davide Cervone (2012): *MathJax: a platform for mathematics on the Web*. Notices of the AMS 59(2), pp. 312–316.

[5] Sébastien Doeraene (2013): *Scala.js: Type-directed interoperability with dynamically typed languages*. Technical Report.

[6] Heather Miller, Philipp Haller, Eugene Burmako & Martin Odersky (2013): *Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization*. In: Acm Sigplan Notices, 48, ACM, pp. 183–202.

[7] Martin Ring & Christoph Lüth (2014): *Collaborative interactive theorem proving with Clide*. In: Interactive Theorem Proving, Springer, pp. 467–482.

[8] Raymond Roestenburg, Rob Bakker & Rob Williams (2015): *Akka in action*. Manning Publications Co.

[9] Carst Tankink (2014): *PIDE for Asynchronous Interaction with Coq*. In: Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, Vienna, Austria, 17th July 2014, *EPTCS* 167, pp. 73–83.

[10] Makarius Wenzel (2014): *System description: Isabelle/jEdit in 2014*. In: Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, Vienna, Austria, 17th July 2014, *EPTCS* 167, pp. 84–94.