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 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 for live changes both by the presenter and the audience.

1 Introduction

Presenting formalized mathematical proofs is a challenge by itself. Formalized proofs, by their very nature, tend to be lengthy, as every 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 crucial 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 is constructed manually; 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, as the resulting view cannot be checked by the prover anymore. 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, and as the whole proof text is shown, the audience is potentially overwhelmed with technical details. 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 us 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.

Autexier, S. and Quaresma, P. (Eds.): Proceedings of 12th
International Workshop on User Interfaces for Theorem Provers (UITP 2016)
EPTCS 239, 2017, pp. 43–52, doi:10.4204/EPTCS.239.4

The paper is structured as follows: Section 2 describes how to work with *Cobra* from a users perspective. Section 3 sketches the architecture of the project, and Section 4 briefly discusses related work. We conclude in Section 5 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 than 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[1], 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. The only strong prerequisite is an active installation of Java 8 or above. To check whether the system setup supports interactive presentation of a specific language, there is the `cobra configure` command which accepts a language identifier as an argument (i.e. `isabelle`, `scala`, `haskell`), which analyses whether the optional prerequisites for the selected language are met and provides further advice.

To create an empty *Cobra* presentation just run `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 to 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*) [1] configuration file. The file `reference.conf` is included in the distribution which contains all available settings together with default values and short descriptions. There is no need to change `cobra.conf` for most presentations. However, among others, the customisations shown in Table 1 are available. Other available settings include the 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`)
`reveal.transition`	the default transition between slides (e.g. `slide`, `fade`, `none`)
`env.isabelle_home`	environment variable to be picked up by language servers

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

[1]The binary can be obtained from `http://www.flatmap.net/cobra` for all major operating systems, or alternatively as source code from `https://github.com/flatmap/cobra/`.

Changing a setting has immediate effect without reloading the application (except for the network interface which can not be switched while cobra is running), such that the user can always observe the effects of the current configuration in the browser.

2.2 Adding slides

The presentation facilities of *Cobra* are based on the *reveal.js* framework [3]. The presentation resides in one HTML file (called `slides.html`). However, this file has no top `html` element (and is thus not valid HTML); instead, the slides are represented by top-level `section` elements and behave as *reveal.js* slides with the exception of `code` elements, which are described in Section 2.4. All the boilerplate needed to make this a valid HTML document, and add the necessary JavaScript and style sheets is automatically generated by *Cobra*.

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

2.3 Integrating LaTeX-style formulae

Text surrounded with dollar signs (i.e. `$a \rightarrow b$`) is interpreted as LaTeX and rendered by *MathJax*, a JavaScript library that can render MathML and a large subset of TeX/ LaTeX and is compatible with all modern browsers [9]. The MathJax configuration can be altered through the `cobra.conf` file.

2.4 Including proofs and code

As mentioned above, `code` elements are treated specially by *Cobra*. By default, every `code` element creates an editable code snippet. By adding certain class attributes to the element, its behaviour can be altered, e.g. the language mode can be defined by adding either `scala`, `haskell` or `isabelle` as a class. When such a class attribute is added, the rich semantic assistance engine is automatically invoked 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 such situations, *Cobra* allows us 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 is 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 extension.

Behavioural Classes can be added to a `code` element to alter its presentation behaviour: `states` enables inline proof states (for Isabelle only), `state-fragments` reveals one proof after another during the presentation, `no-infos` hides info messages during the presentation, and `no-warnings` hides warning messages during the presentation.

```html
<code class="hidden" src="src/Seq.thy">
</code>
<section>
  <h2>A Short Demo</h2>
  Sequences and their concatenation
  <code src="#def-seq-conc">
  </code>
</section>
<section>
  <h2>A Short Lemma</h2>
  <code src="#reverse-conc" class="states">
  </code>
</section>
<section>
  <h2>A Short Proof</h2>
  <code src="#reverse-reverse" class="states">
  </code>
</section>
<section>
  <h2>A Short Haskell Demo</h2>
  <code class="haskell">
    module Example where
    fibs = {-(-}undefined{-|0 : 1 : zipWith (+)
        fibs (tail fibs))-}
  </code>
</section>
<section>
  <h2>A Short Scala Demo</h2>
  <code class="scala">
    object Example {
      val x = /*(???|*/3 * 7/*)*/
    }
  </code>
</section>
```

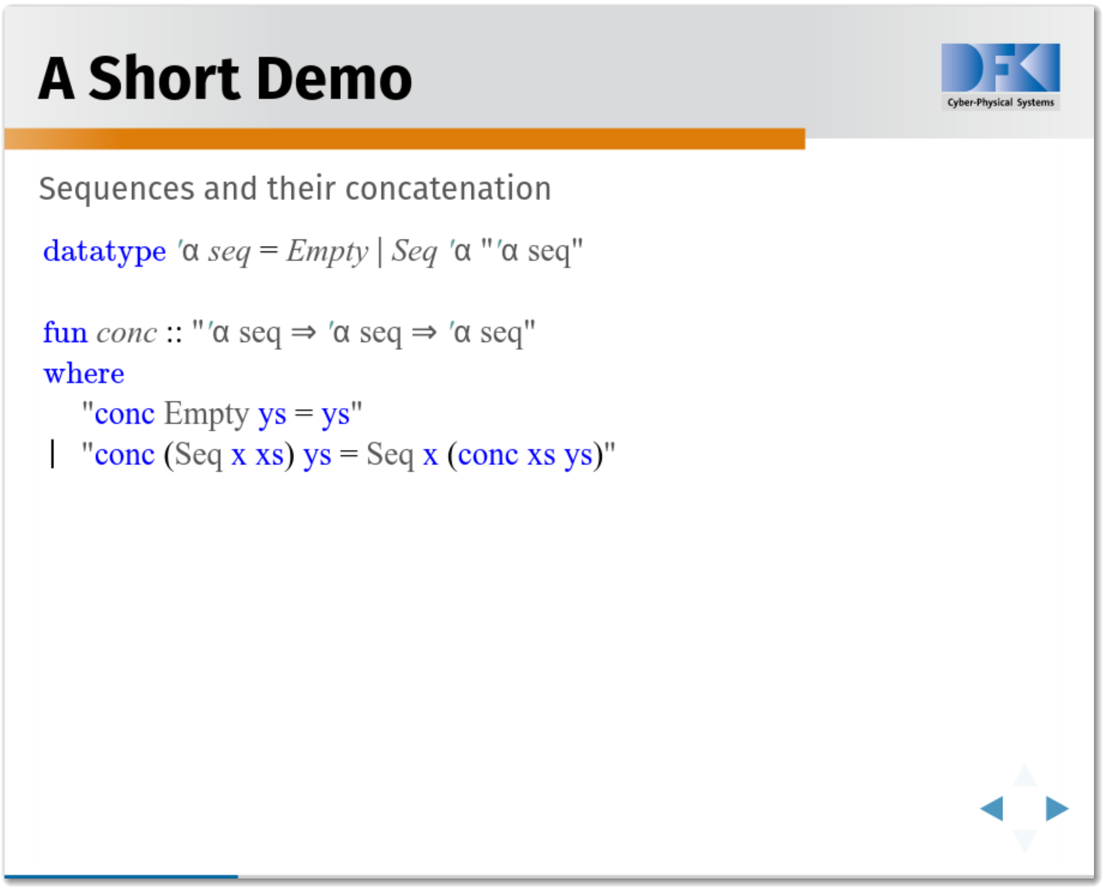
(a) slides.html

```
theory Seq
imports Main
begin

(** begin #def-seq-conc *)
datatype 'α seq = Empty | Seq 'α 'α seq

fun conc :: 'α seq ⇒ 'α seq ⇒ 'α seq
where
  conc Empty ys       = ys
| conc (Seq x xs) ys  = Seq x (conc xs ys)
(** end #def-seq-conc *)

fun reverse :: 'α seq ⇒ 'α seq
where
  reverse Empty       = Empty
| reverse (Seq x xs)  = conc (reverse xs) (Seq x Empty)

lemma conc_empty: conc xs Empty = xs
  by (induct xs, simp_all)

lemma conc_assoc: conc (conc xs ys) zs = conc xs (conc ys zs)
  by (induct xs, simp_all)

(** begin #reverse-conc *)
lemma reverse_conc:
    reverse (conc xs ys) = conc (reverse ys) (reverse xs)
  apply (induct xs)
  apply (simp_all add: conc_empty conc_assoc)
  done
(** end #reverse-conc *)

(** begin #reverse-reverse *)
lemma reverse_reverse: reverse (reverse xs) = xs
  oops
(** end #reverse-reverse *)

end
```

(b) src/Seq.thy

Figure 1: Simple example with five slides and external Isabelle source.

Code Fragments are parts of code that are revealed or altered interactively during the presentation. There is a special comment syntax which preserves the semantics of the original source file. The variants of a section of source code are enclosed in parentheses and separated by a pipe symbol. The parentheses are then enclosed in block comments, including either the left or right variant. e.g. `val x = /*(*/??? /*|3 * 7)*/` or `val x = /*(???|*/3 * 7/*)*/`, which will both result in the same sequence during the presentation but obviously produce different meanings in the source file. When no alternative is provided (as in **lemma** *x*: *A* ⇒ `(*(*)`*A*`(*)*)`) the code fragment is selected during the presentation and semantic information is annotated if available.


A Short Demo

Sequences and their concatenation

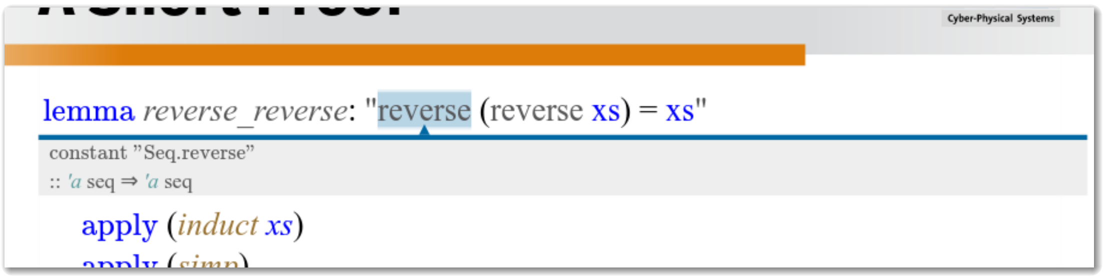
```
datatype 'α seq = Empty | Seq 'α "'α seq"

fun conc :: "'α seq ⇒ 'α seq ⇒ 'α seq"
where
   "conc Empty ys = ys"
|  "conc (Seq x xs) ys = Seq x (conc xs ys)"
```

Figure 2: The rendered result of Listing 1a. The syntax highlighting for the inner syntax (i.e. the double quoted parts) is provided by the prover.

2.5 Interactive presentations

Use `cobra <dir>` on the command line to start a presentation server within the specified directory. After a short while *Cobra* will be initialised. The presentation can now be opened with any modern web browser. The user will see the first slide. It is possible to navigate through the slides with the keyboard or a presenter. *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 semantic information (Figure 2). The syntax highlighting will reflect the semantic meaning of tokens, just like in Isabelle/jEdit [20] or Clide [16, 13]. 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 a red background behind the affected part of the snippet (Figure 4), which is useful when presenting a situation where an error is of importance (we found the usual red underlining to be too innocuous). It is also possible to view additional information about entities by selecting 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

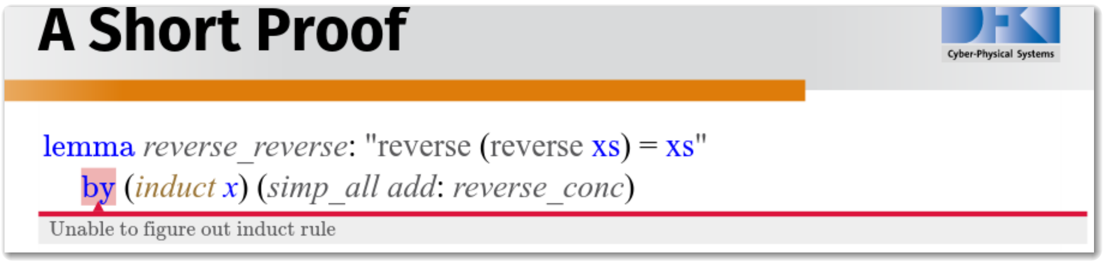

Cyber-Physical Systems

lemma *reverse_reverse*: "reverse (reverse xs) = xs"

constant "Seq.reverse"
:: 'a seq ⇒ 'a seq

apply (*induct xs*)

Figure 3: A hover tool-tip displays information about “reverse”


A Short Proof

Cyber-Physical Systems

lemma *reverse_reverse*: "reverse (reverse xs) = xs"
by (*induct x*) (*simp_all add*: *reverse_conc*)

Unable to figure out induct rule

Figure 4: An error is displayed

Isabelle/jEdit (Figure 3). Since manually selecting can be cumbersome it is also possible to use *selection fragments*, which automate this by allowing to step through predefined ranges as described in Section 2.4.

Furthermore, it is possible to alter the content of snippets just like in a text editor. [2] When snippets 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, if the speaker allows access to the presentation server[3]. The changes will then be synchronised to everybody else and become visible in the main presentation. 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.

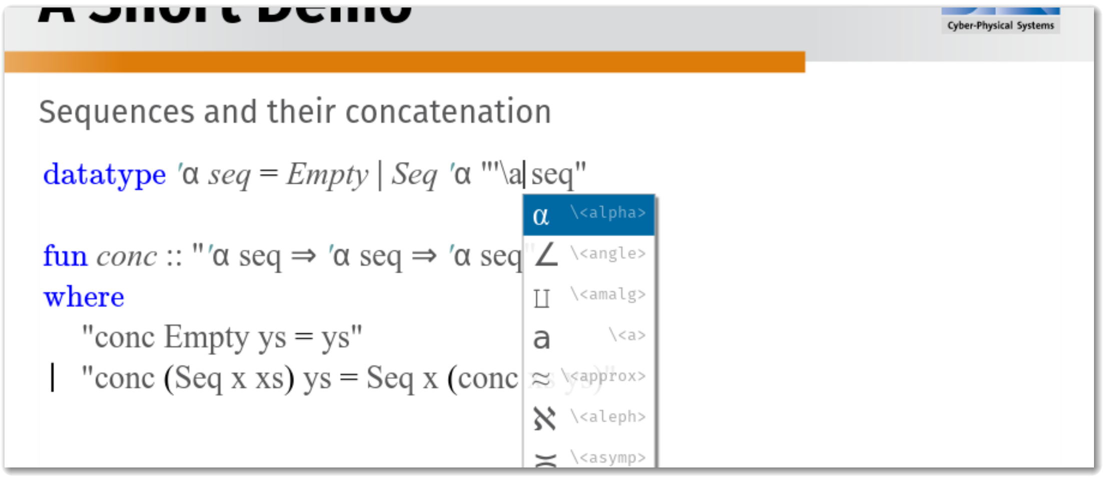
2.6 Publishing and distribution

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.

[2]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.

[3]The default setting of serving to `localhost` prevents this.


Sequences and their concatenation

```
datatype 'α seq = Empty | Seq 'α "'\a seq"

fun conc :: "'α seq ⇒ 'α seq ⇒ 'α seq
where
   "conc Empty ys = ys"
|  "conc (Seq x xs) ys = Seq x (conc
```

Figure 5: Editing the proof during the presentation

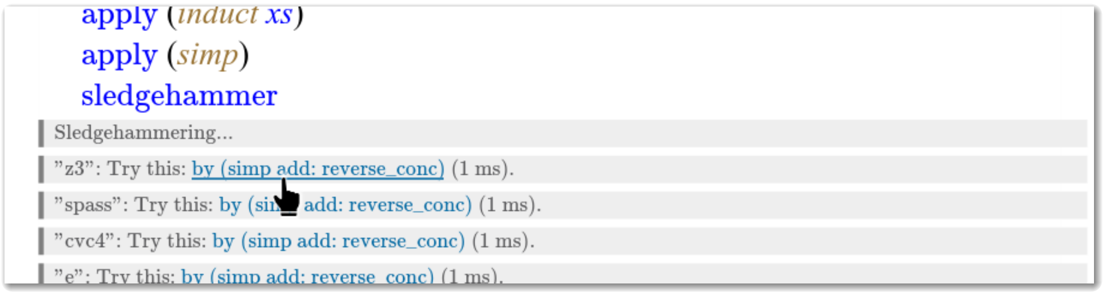

```
  apply (induct xs)
  apply (simp)
  sledgehammer
Sledgehammering...
"z3": Try this: by (simp add: reverse_conc) (1 ms).
"spass": Try this: by (simp add: reverse_conc) (1 ms).
"cvc4": Try this: by (simp add: reverse_conc) (1 ms).
"e": Try this: by (simp add: reverse_conc) (1 ms).
```

Figure 6: Using a result from an asynchronous task

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 correct rendering of snippets for printing.

3 *Cobra* internals

The PIDE framework [20] which was developed together with the Isabelle/jEdit integration has already been successfully integrated into the collaborative web environment *Clide* in previous work [16]. We were able to reuse significant portions of the code to implement *Cobra*. However, due to the limitations at the time, while the *Clide* server is implemented in Scala, a strongly typed and reasonably well specified language, the client was implemented in CoffeeScript, a thin wrapper language compiling directly to JavaScript. Both CoffeScript and JavaScript 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.

Recently, several independent approaches, like TypeScript [6], arose to remedy these shortcomings. A particularly interesting solution in our context is *ScalaJS* [10], 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.

By using *ScalaJS* the *Cobra* code base does not need to 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 [17], a minimal HTTP library based on the Akka implementation of reactive streams [2]. 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 [14]. This allows us 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 [16]).

4 Related Work

There is a wealth of work on presenting formalised proof, going back right to the start of the field. Early attempts were concerned with making proof scripts more like mathematical texts rather than programming languages; one of the earliest representatives of this was the Mizar prover. However, this does not allow interaction with the proof script beyond it being checked by the prover. There have been various attempts at presenting interactive views on proofs, most of which focus on specific aspects, logics or systems: the Jape prover [7] allows direct manipulation of a natural deduction proof visualised as a tree or box, Grundy and Back use structured calculational reasoning [11, 5] where the proof can be interactively explored along its hierarchical structure, the Omega system visualised proofs interactively using proof trees [18], Theorema uses the computer algebra system Mathematica [8], and there have been various attempts to visualise geometric proofs using diagrams [21, 15, 22].

On a less conceptual and more technical level, there are systems which are technologically similar to *Cobra* in that they make use of web-based front-end technology: ProofWeb was an early precursor [12], Clide offers a full web interface for PIDE-enabled provers [16], and jsCoq moves the whole prover into the web browser by translating it into JavaScript [4]. However, while these are similar in some regards, differences remain; *e.g.* jsCoq is completely devoted to Coq, and not generic like *Cobra*.

Summing up, the following three characteristics distinguish *Cobra* from other systems: firstly, it focuses on presentations (slides) as opposed to long-form text, which means that extraction of parts of the proof plays a key role; secondly, it is interactive, allowing to change content and display the results immediately; and thirdly, it is generic, *i.e.* usable with different provers or programming languages.

5 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. Our tool is ready for use in production; it is distributed in binary form for all major operating systems at `http://www.flatmap.net/cobra`[4], or alternatively in source code form at `https://github.com/flatmap/cobra/`.

We envisage the following usage 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 on 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 [19], 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] *HOCON Specification*. `https://github.com/typesafehub/config/blob/master/HOCON.md`. Accessed: 2016-09-20.

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

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

[4] Emilio Jesus Gallego Arias, Benoit Pin & Pierre Jouvelo (2016): *jsCoq: Towards a Hybrid Theorem Proving Interface for Coq*. In Serge Autexier & Pedro Quaresma, editors: *Proc. User Interfaces for Theorem Provers (UITP'16)*, Coimbra.

[5] Ralph Back, Jim Grundy & Joakim von Wright (1997): *Structured calculational proof*. *Formal Aspects of Computing* 9(5), pp. 469–483, doi:10.1007/BF01211456.

[6] Gavin Bierman, Martin Abadi & Mads Torgersen (2014): *Understanding typescript*. In: *ECOOP 2014–Object-Oriented Programming*, Springer, pp. 257–281, doi:10.1007/978-3-662-44202-9_11.

[7] Richard Bornat & Bernard Sufrin (1999): *Animating Formal Proof at the Surface: the Jape proof calculator*. *The Computer Journal* 42(3), pp. 177– 192, doi:10.1093/comjnl/42.3.177.

[4]Please note you need to have Java 8 installed to run the *Cobra* binary.

[8] Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky & Wolfgang Windsteiger (2016): *Theorema 2.0: Computer-Assisted Natural-Style Mathematics*. Journal of Formalized Reasoning 9(1), pp. 149–185, doi:10.6092/issn.1972-5787/4568. Available at `https://jfr.unibo.it/article/view/4568`.

[9] Davide Cervone (2012): *MathJax: a platform for mathematics on the Web*. Notices of the AMS 59(2), pp. 312–316, doi:10.1090/noti794.

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

[11] Jim Grundy (1996): *A Browsable Format for Proof Presentation*. Technical Report 22, Turku Centre for Computer Science.

[12] Cezary Kaliszyk (2007): *Web Interfaces for Proof Assistants*. In Serge Autexier & Chris Benzmüller, editors: Proc. User Interfaces for Theorem Provers (UITP'06), Electronic Notes in Theoretical Computer Science (ENTCS) 174, pp. 49– 61, doi:10.1016/j.entcs.2006.09.021.

[13] Christoph Lüth & Martin Ring (2013): *A Web Interface for Isabelle: The Next Generation*. In: Intelligent Computer Mathematics, Springer, pp. 326–329, doi:10.1007/978-3-642-39320-4_22.

[14] 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, doi:10.1145/2509136.2509547.

[15] Julien Narboux (2007): *A Graphical User Interface for Formal Proofs in Geometry*. Journal of Automated Reasoning 39(2), pp. 161–180, doi:10.1007/s10817-007-9071-4.

[16] Martin Ring & Christoph Lüth (2014): *Collaborative interactive theorem proving with Clide*. In: Interactive Theorem Proving, Springer, pp. 467–482, doi:10.1007/978-3-319-08970-6_30.

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

[18] Jörg Siekmann, Stephan Hess, Christoph Benzmüller, Lassaad Cheikhrouhou, Armin Fiedler, Helmut Horacek, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Martin Pollet & Volker Sorge (1999): *LOUI: Lovely OMEGA User Interface*. Formal Aspects of Computing 11(3), pp. 326–342, doi:10.1007/s001650050053. Available at `http://christoph-benzmueller.de/papers/J2.pdf`.

[19] 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, doi:10.4204/EPTCS.167.9.

[20] 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, doi:10.4204/EPTCS.167.10.

[21] Sean Wilson & Jacques D Fleuriot (2005): *Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs*. In: Workshop on User Interfaces for Theorem Provers (UITP). Available at `https://www.inf.ed.ac.uk/publications/online/0242.pdf`.

[22] Zheng Ye, Shang-Ching Chou & Xiao-Shan Gao (2010): *Visually Dynamic Presentation of Proofs in Plane Geometry*. Journal of Automated Reasoning 45(3), pp. 213–241, doi:10.1007/s10817-009-9162-5.