Web Interfaces for Proof Assistants

Cezary Kaliszyk [1]

Radboud University Nijmegen, The Netherlands

Abstract

This article describes an architecture for creating responsive web interfaces for proof assistants. The architecture combines current web development technologies with the functionality of local prover interfaces, to create an interface that is available completely within a web browser, but resembles and behaves like a local one. Security, availability and efficiency issues of the proposed solution are described. A prototype implementation of a web interface for the Coq proof assistant [8] created according to our architecture is presented. Access to the prototype is available on `http://hair-dryer.cs.ru.nl:1024/`.

Key words: Proof Assistant, Interface, Web, Coq, Asynchronous DOM modification

1 Introduction

1.1 Motivation

Nowadays people are more and more accustomed to having a connection to the Internet all the time. Thus the network becomes a part of the computer one uses. As a consequence a tendency has emerged to provide services available just by accessing certain web pages. In this way people do not themselves need to install software for such services on their computers any more. Examples include web interfaces to e-mail, calendars, chat clients, word processors and maps.

Commercial services are often available through web-interfaces. On the other hand, in the scientific domain, examples are not so abundant. In particular there are no real implementations of web interfaces for proof assistants.

To use a proof assistant, one needs to install some software. Often the installation process is complicated. For example to install Isabelle [17], which is one of the most popular proof assistants, on a Linux system, one needs a particular version of PolyML, a HOL heap and Isabelle itself. To use an

[1] Email: `cek@cs.ru.nl`

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

interface to access the prover, one needs ProofGeneral [4] and one of the supported Emacs versions. With Debian we had to downgrade the Linux kernel to support PolyML. The process described above is already complicated, not to mention other operating systems and architectures, additional desirable patches and libraries, or less commonly used provers.

This is a problem. It happens that computer scientists prefer to stick with installed old versions of provers, not to go through the same process to upgrade. Mathematicians may even stay away from computer assisted proving, just because of the complexity of installation.

We want a fast interface, that is available with just a web browser. We want to access various proof assistants and their versions, in a uniform manner, without installing anything, not even plugins. The interface should look and behave like local interfaces to proof assistants.

We want the possibility to create web pages, that show tutorials and proofs, but that are bound to the prover itself, where the user can interact with the real system. The provider of the server may install patched versions of provers, allowing an easy way for the users to try out their features. We want libraries for proof assistants to be available centrally, so that users who want to see them do not need to download or install anything. The interface should allow developing proofs and libraries centrally, in a *wiki*-like [11] way.

1.2 Our Approach

The solution is a client-server architecture with a minimal lightweight client interpreted by the browser, a specialized HTTP server and background HTTP based communication between them. The key element of our architecture is the asynchronous DOM modification technique (sometimes referred to as *AJAX - Asynchronous JavaScript and XML* or *Web application*). The client part is on the server, and when the user accesses the interface page, it is downloaded by the browser, which is able to interpret it without any installation.

The user of the interface, accessing it with the browser, does not need to do anything when a modification is done on the server. Every time the user accesses a prover, the version of the prover that is currently installed on the server is used. The user can access any of the provers installed on the server, even a prover which does not work on the platform from which the connection is made.

Saving the files on the central server allows accessing them from any location, by just accessing the interface's page with a web browser. A central repository simplifies cooperation in proof development, by replacing versioning systems like CVS, which keeps a remote and a local copy, by a *wiki-like* mechanism, where the only copy is the remote one.

Our approach is presented as an architecture to create web interfaces to proof assistants, but it is not limited to them. The problems solved are relevant to creating web interfaces programs that have a state, include an *undo*

mechanism, and their interfaces can be buffer oriented. Our architecture may be applied for example to buffer oriented programming languages, like Epigram [15].

1.3 Related work

There have been some experiments with providing remote access to a prover. None of them allowed efficient access without installing additional software.

LogiCoq [18] is a web interface to Coq [8]. It offers a window where one can insert the contents of whole Coq buffer and submit them for verification. It sends the whole buffer with standard HTTP request and refreshes the whole page. Therefore one can work efficiently only with tiny proofs.

The web interface to the Omega system [7], requires the Mozart interpreter to be installed on the user's machine. The use of the web browser is minimal, the whole interface is written in Mozart. Installation of Mozart is possible only for certain platforms which also makes the solution limited.

There are Java applets having built-in proof assistant functionality. Examples may include G4IP [19] or Logic Gateway [12]. The installation of a browser plug-in to support Java is not simple in a Unix environment and limiting provers to Java applets is undesired.

Web interfaces related to proof assistants and displaying mathematics on the web are worth mentioning. In particular:

- Helm [3] - (Hypertextual Electronic Library of Mathematics) A web interface that allows visualisation of libraries available for proof assistants.
- Whelp [2] - A content based search engine for finding theorems in proof assistants libraries, that supports queries requiring matching and/or typing.
- ActiveMath [16] - A web-based framework for learning mathematics that uses Java applets to communicate with a central server using OMDoc [13].

There are some commercial web interfaces and frameworks that use asynchronous DOM modification in non scientific domains.

The novelty of our architecture in comparison with existing web interfaces for theorem provers is that it allows the creation of an interface to a prover, that can look and behave very much like the ones offered by state-of-the-art local interfaces, but is available just by accessing a page with a web browser without installing any additional software, not even plugins. Because of the architecture, the network used to transfer information does not slow down the interaction. The idea to use asynchronous DOM modification to create an interface to a proof assistants has never been applied before.

1.4 Contents

In the rest of the paper we present the techniques for creation of web interfaces, that we will use (Section 2) and the internals of a local prover interfaces

which we try to imitate (Section 3), followed by the presentation of the new architecture (Section 4) and a description of its security and efficiency (Section 5). We present our implementation prototype (Section 6). Finally we conclude and present a vision of future work (Section 7).

2 The Concept of Asynchronous DOM Modification

As the web is becoming more commonly used, web page designers and browser implementers add new functionality to web pages. Text files have been replaced by hyper-linked files, later including images, language-specific and mathematical characters, styles and dynamic elements. The W3C Consortium, which is the organization responsible for the standardization of the Web, defines these elements as standards, and in consequence they are implemented in a similar ways in all browsers.

Since the late nineties browsers have started supporting the following technologies relevant to our research: JavaScript, DOM [14] and XmlHttp [20]. Combined use of these three technologies has became popular in recent years, since they allow one to create responsive web interfaces. In this document we refer to the combined usage of these three technologies as "*Asynchronous DOM Modification*." One can find other names describing this technique, like *AJAX* or *Web Application*.

JavaScript is a scripting programming language, created by Netscape in 1995, for adding certain dynamic functionality to pages written in HTML. It has been quickly adopted by most browsers and nowadays it is supported even by some text mode browsers like w3m and Links, and mobile phone browsers. It is very often used on Internet websites.

DOM (Document Object Model) [14] is an API (Application programming interface) for managing HTML and XML documents that allows modifications of their structure and content. Recent browsers support W3C DOM accessibility by JavaScript. It is often used on web pages to add dynamic elements, for example drop-down menus or images that change when the mouse moves over them.

XmlHttp [20] is an API accessible by web browser scripting languages to transfer data to and from a web server. It internally uses HTTP requests. XmlHttp requests are sent to the server without the knowledge of the user of the web browser. For every XmlHttp request a callback has to be provided, to be executed when the response from the server is received. The sending of the request can be optionally asynchronous. XmlHttp has been available in most browsers for some time, and has been recently described in a W3C specification draft.

Asynchronous DOM modification is a web development technique that uses the three technologies described above to create responsive web interfaces. Such interfaces are web pages, where particular events (key presses and mouse movement) are captured by JavaScript events. The minimal client

part encoded in JavaScript processes the local events, like menu opening or typing in a buffer. Events that require additional information from the server are sent in asynchronous XmlHttp requests. Since the request is done in the background, it does not interrupt the user from working locally. When the response arrives, it is used to modify the DOM of the page.

In comparison with classical web pages, the usage of asynchronous DOM modification makes it possible to send minimal information to the server, to receive only the information required, and to refresh small parts of the web page. Network overhead and page refreshing are minimized, thus creating interfaces which work many times faster then classical web-based ones. This way, the interface can closely resemble local interfaces, if network latency is reasonable. In case of high network latency, asynchronous requests allow the user to work locally, while additional data is requested.

Examples of usage of the asynchronous DOM modification are: webmails and calendars which operate within a single page, maps which download required parts as they are dragged, and web chat clients. Such web interfaces are supported by all standard web browsers, in particular all Gecko based browsers, Microsoft Internet Explorer versions from 5, Opera from version 8, Konqueror from version 3.2, Safari from version 1.2 and even Nokia S60 browser from version 3. It is not supported by text mode browsers and browsers for visually impaired people.

3 Generic Interface for Proof Assistants

In this section we describe the internals of local interfaces for proof assistants. We chose for this ProofGeneral [4] for two reasons. First, it is a prover-independent interface to proof assistants. Second, it is popular, since it is universal and since it is built on the highly configurable Emacs text editor.

ProofGeneral's interface provides the user with two buffers: an editable buffer containing the proof script and the prover state buffer. ProofGeneral relies on the proof assistant to process the commands incrementally. It does not distinguish tactic-mode proofs from declarative-mode proofs. State changing and non-state-changing Coq commands are distinguished to make only the relevant ones part of the proof script and to allow queries.

The interface colors keywords according to the above distinctions, and additionally marks parts of the buffer with a background color, to indicate the status of verification. Possible states include: Expression that has been accepted by the prover, expression that is now being verified, and editable non-verified expression.

ProofGeneral provides a proof replying mechanism. The prover itself has to provide an *undo* mechanism. Users may choose a point in the buffer to go to, and ProofGeneral issues a number of proof steps and *undo* steps to the prover in order to reach that point.

ProofGeneral is responsible for providing the proof script from files on

the disc to the prover and saving the buffers state. Other disc operations that exist in some provers, like proof compilation, program extraction or automated creation of documentation are not handled by ProofGeneral.

The current version of ProofGeneral is implemented mostly in Emacs Lisp, and is strongly tied with the editor itself. It is easy to adapt ProofGeneral to new proof assistants, by setting a number of variables. If this is not sufficient ELisp code can be used.

Other interfaces to provers offer mostly similar functionality. In some interfaces, like PCoq [1] or IsaWin, additional visualisation mechanisms are available, for example *term annotations*. Some of these mechanisms are not available in ProofGeneral; this limitation comes from the Emacs editor.

4 General Architecture

The two core elements of our architecture are: a specialized Web server and a communication mechanism (Fig. 1).

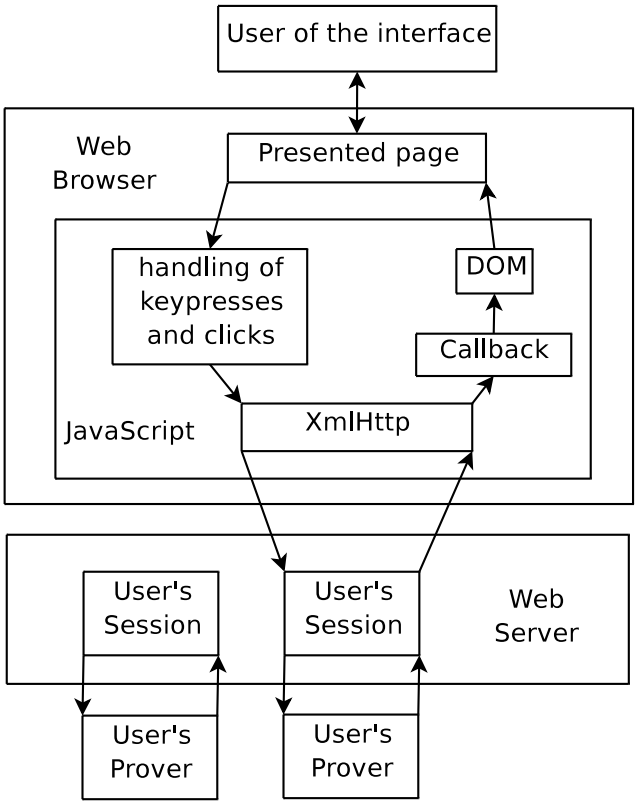

Fig. 1. General architecture.

The Web server serves normal files and it is able to respond to special HTTP requests (see 4.2). The main interface is available as a normal HTML file on the server. When a user accesses the page with a browser, the page requires certain JavaScript files, which are then downloaded and interpreted by the browser. This serves as the client part.

The communication between the client part and the server is done with the mechanism described in Section 2. HTTP requests are created in the background. The results are used to update the page in place. Only a small amount of information is transferred between the client and the server. The transfer is done asynchronously, making the interface responsive.

4.1 The Client Part

The client part offers a web page that initially presents the user with an editable buffer and an empty response buffer. (Also a menu or a toolbar is necessary for interaction, but they are normal elements of web pages). Buffers are implemented as HTML IFrames[2]. All keys that modify the IFrame are assigned to a special function. Locking of parts of the buffer is implemented by disallowing changes to locked parts of the buffer in this function.

When the user wants to verify a part of the buffer, this part is locked and sent to the server. Since the request is a background one, even if it takes a moment the user may continue working. When the response arrives, the contents of the two buffers are modified. The response may be a success, and then the part of the editable buffer is marked 'verified' and the response buffer shows the new prover state. If the command failed, the part of editable buffer is unlocked and the error shown. Parts of the editable buffer are marked, as their state changes, by using background colors, as it is done in ProofGeneral.

The interface includes a proof replaying mechanism, created in a similar way that it is done in local interfaces. When the user wants to go to a particular place in the buffer, this information is passed to the server. The server sends the commands to the client's prover session and informs the interface about the results. In a similar way the interface includes a *break* mechanism that allows stopping the prover's computation.

The interface includes functionality for file interaction. Files can be loaded and saved on the server. For interoperability downloading files and uploading files from the local computer may be provided. For proof development efficiency, insertion of templates and queries may be provided.

4.2 The Server Part

The server includes standard HTTP file serving functionality. With it the user's browser downloads the client part. The server can also handle special messages available for users, that have logged in. Session mechanism is used to support multiple clients. A session is created when a user logs in to the system and is sustained with a cookie mechanism. Every user's session is associated with a particular prover session. The server runs provers as subprocesses and communicates with them through standard input and output. Prover sessions

[2] An *IFrame* is an HTML tag that includes a floating frame within a page, that can be optionally editable.

are terminated after a long period of inactivity (if the user did not close the page, the client part can replay the proof script from the beginning).

The special messages, mentioned above, include: passing a given complete expression to verify to the prover, issuing an *undo* command in the prover, saving a file, loading a file, and *break* (stopping the prover computation). The commands from the client for the prover are passed first to the server, which transmits them to the prover. Prover replies are analysed by the server and only state changes are sent to the client. The state changes consist of two parts: changing of the markings of the edit buffer and the new contents of the prover state buffer.

Replies from the server are passed back to the client in an asynchronous way. This means, that the server does not answer HTTP requests from the client immediately, but when an answer from the prover is received or a time-out is reached. The server keeps a pool of provers that have been asked to process data, and waits for an answer from any of them. The waiting process does not block the server, that is, other clients' requests can be processed in the meantime.

5 Security and Efficiency

5.1 User side

All code that the user runs is interpreted within the web browser. Thus a malicious or virus infected prover can influence the client only by exploiting system or browser errors.

The efficiency of code execution on the user's side is dependent on the efficiency of the browser's internal web page and scripts interpretation, and the speed of HTML rendering.

Our experiments show that client-side DOM changes with Internet Explorer are approximately twice as fast as with Mozilla Firefox (still usually invisible for the user). It is hard to say whether this is due to less security checks or the worse quality of the rendered page (no anti-aliasing) in Internet Explorer.

5.2 Server side

In any centralized environment security, availability and efficiency of the server are important. Standard security measures include a backup server prepared to take over network traffic in case of a primary server failure and regular backing up of user files. In this subsection we will describe only the issues and solutions particular to a server that runs a web interface to a prover.

Three kinds of issues arise: security, availability and equal sharing of resources. First, exploiting bugs in our architecture could lead crackers to take control of the server. Second, in a centralized environment the only copy of files is on the server. Unavailability of the server makes users not only unable

to work, but also unable to access their files. Last, when users access the same server its resources are shared. If a particular prover uses all the memory or CPU, other users are unable to work.

To provide security, the server is run in a chrooted[3] environment, as a non-privileged user. The permissions include only reading server files and executing the provers. Every prover type is run as a different user (using the file `setuid` mechanism), that has read rights only to the prover's library, and write rights only to a directory where the prover's proof scripts are stored. To disallow storing overly large amounts of data, filesystem quota may be used.

For provers that allow system interaction, this functionality can be sometimes disabled. In particular, for ML based provers, dropping to the toplevel can be disabled. If the server administrator doesn't trust the prover's implementation, a secure version of the kernel can be used to disable irrelevant system calls. In this case even a language that is implemented inside ML can be available without changes to the prover itself.

To ensure equal sharing of resources, prover processes can be run with Cpu quota and memory quota mechanisms. The scheduling policy can be changed (for example with the `nice` system call) to provide the server process with priority over prover processes. Different provers have different CPU and memory requirements, which should be taken into account while setting the limits.

When many users want to access the interface, the resources of a single server may be insufficient. It is simple to run the server on a set of machines, by calling provers as subprocesses through `ssh` on separate computers. A load balancing mechanism can be implemented.

The communication between the server part and the client part can be secured by providing the interface through HTTPS.

6 Implementation of a Prototype

We have implemented a minimal prototype of a web interface that follows the proposed architecture. The interface allows using the Coq proof assistant [8] with just a web browser, but it looks and behaves (Fig. 2) like the ones offered by CoqIde and ProofGeneral.

Our server is a 400 line OCaml program, that serves two HTML files and a number of JavaScript files. It additionally supports special POST requests for verifying and for undoing commands as well as for loading and saving of files. It uses the OCamlHttpd library, for web-server functionality.

Our client consists of 10`kB` of JavaScript and 2`kB` of HTML. Most of the client-side code is responsible for the locking of the buffer and recognition of Coq expressions.

[3] `chroot` is a system call preventing a process to access any files outside of a special root directory.

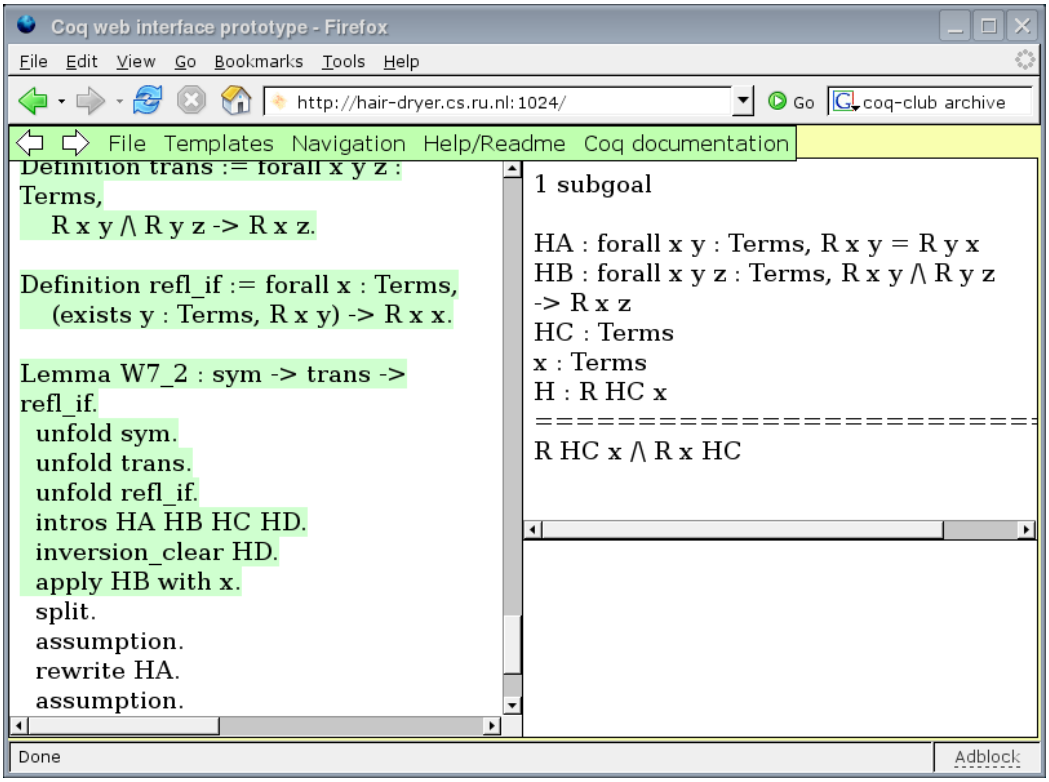

Fig. 2. Screenshot of the prototype, that shows working with a Coq proof. The verified part of the edit buffer is colored and locked. The state buffer shows the state of the proof, there are no Coq warnings.

To secure our prototype the server is run as `nobody` in a minimal `chroot`ed environment. The prover sub-processes are `reniced` not to interfere with the main server process. Dropping from Coq to OCaml toplevel is disabled. The access to the interface is password protected, to avoid creating prover sessions for web-spiders. Web spiders are able only to see the saved proof scripts.

Our prototype includes a `1kB` file, that is supposed to create a uniform layer that works with different browsers. We have not yet made it as general, as the asynchronous DOM modification is. In particular our prototype works well with Gecko-based browsers (Mozilla, Firefox, Galeon, ...). It works with Internet Explorer 6 and Opera 9, with some key-bindings missing (these browsers have them assigned to internal functions). It does not yet work with KHTML based browsers (Konqueror, Safari) and older versions of the above. We have tested our implementation's efficiency, by trying to use the server from other locations. Although measuring responsiveness to user's actions is hard to be done objectively, our experiments show, that with reasonable network latency, its responsiveness is very good.

The prototype is a Coq web interface, but there is not much code specific to Coq. The client part includes recognition of Coq comments and whole expressions to send. The server part includes recognition of successes and failures as well as the *undo* mechanism. For all ELisp code from ProofGen-

eral equivalent JavaScript regular expression handling can be provided. Thus adapting these three things to other provers should be simple, which is why we believe that implementing an interface according to our architecture that would support different provers can be easily done.

The client part has to overcome the minor differences between browsers. In particular it includes functions that create a uniform layer for XmlHttpRequest creation, event binding, and DOM that work the same way on all currently supported browsers.

6.1 Possible Uses

Our interface can be used to create interactive tutorials presenting proof assistants. We have created a special proof script, that includes a slightly modified version of the official Coq tutorial. The descriptive parts have been put inside comments (including the HTML formatting), and commands to the proof assistant have been left outside comments. A user that enters such a page may just read the tutorial and execute the commands in Coq environment, but may also do own experiments with it.

Non-trivial proof scripts that use tactics are unreadable without intermediate proof states. Thus proofs presented on the web are usually accompanied with some of the proof states usually automatically generated by Coqdoc or TeXmacs [6]. A web interface can be used (even in a read-only mode) to present such proofs interactively. In this way, the user reading the proof chooses which proof states to see.

External proof assistant libraries can be included on the server. With our server we included C-CoRN (Constructive Coq Repository at Nijmegen) [10]. Such libraries can be developed on the server. In such an approach visitors can always see and test the current version, without downloading and compiling the library.

Modified and experimental versions of provers usually require patching a particular version of the source of the proof assistant. Presenting such a modified version to others is easily possible with the given infrastructure. The server offered includes the Declarative Proof Language extension for Coq [9].

7 Conclusion

We presented an architecture to create simple, lightweight and fast web interfaces to proof assistants. Such interfaces are a novelty in the domain. Our solution works with modern web browsers without installing any additional software. The installation and updating process is done only on the server, the users do not need to do anything. It is therefore completely platform independent.

The communication mechanism makes the usage of the network minimal, therefore making the interface comparably responsive to local ones. In com-

parison with other client-server solutions, the only limitation is the dependency on the web browser. Fortunately web browsers include full scripting languages, allowing implementation of nearly all possible functionality of the interface on the client side. In particular the browser's internal editors are weak in comparison with local editors. One can implement in JavaScript the handling of more key bindings to make the editor similar to a local one. Most features of state-of-the-art local interfaces for proof assistants can be imitated this way. The efficiency of an editor implemented in JavaScript would depend on the browser interpreting it. We have not been able to find any such editor.

We believe that a centralized environment, with provers accessible through a web interface, is not limited in comparison with local interfaces, and that the architecture we have proposed is in the spirit of the current trends of development in computer science.

7.1 Future Work

Our primary focus is to extend the proposed architecture to a complete wiki-like architecture. This requires a versioning mechanism and merging of users' changes on the server. Additionally proof displaying and searching mechanisms are mandatory. Editing conflicts can be resolved in similar way as it is done in wiki software. For example if the file was changed and a user wants to save over it, differences are presented.

We would like to see how well our solution fits with the general prover interaction protocol PGIP [5]. The protocol is XML-based, so parts of it may even be passed by the server directly to browsers, since they are already able to parse XML. On the other hand the protocol may include too much information, since it was designed as a local one.

Finally we would like to create an implementation that includes all the features of our proposed architecture. Ideas include: providing other provers, making it compatible with all browsers that support asynchronous DOM modifications, implementing the *break* mechanism, compiling Coq files automatically, adding syntax highlighting, and providing better security.

References

[1] Amerkad, A., Y. Bertot, L. Pottier and L. Rideau, *Mathematics and proof presentation in PCoq*, Rapport de Recherche 4313, Inria, Sophia Antipolis (2001).

[2] Asperti, A., F. Guidi, C. S. Coen, E. Tassi and S. Zacchiroli, *A content based mathematical search engine: Whelp.*, in: J.-C. Filliâtre, C. Paulin-Mohring and B. Werner, editors, *TYPES*, Lecture Notes in Computer Science **3839** (2004), pp. 17–32, URL: http://www.bononia.it/~zack/stuff/whelp.pdf.

[3] Asperti, A., L. Padovani, C. S. Coen and I. Schena, *Helm and the semantic math-web.*, in: R. J. Boulton and P. B. Jackson, editors, *TPHOLs*, Lecture

Notes in Computer Science **2152** (2001), pp. 59–74,
URL: http://helm.cs.unibo.it/smweb.ps.gz.

[4] Aspinall, D., *Proof General: A generic tool for proof development.*, in: S. Graf and M. I. Schwartzbach, editors, *TACAS*, Lecture Notes in Computer Science **1785** (2000), pp. 38–42.

[5] Aspinall, D., C. Lüth and D. Winterstein, *A framework for interactive proof*, in: D. Aspinall, editor, *Proceedings of the ETAPS-05 Workshop on User Interfaces for Theorem Provers (UITP-05), Edinburgh*, 2005, p. 15,
URL: http://proofgeneral.inf.ed.ac.uk/Kit/docs/pgipimp.pdf.

[6] Audebaud, P. and L. Rideau, *Texmacs as authoring tool for formal developments.*, Electr. Notes Theor. Comput. Sci. **103** (2004), pp. 27–48.

[7] Benzmüller, C., L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, A. Meier, E. Melis, W. Schaarschmidt, J. H. Siekmann and V. Sorge, *Omega: Towards a mathematical assistant.*, in: W. McCune, editor, *CADE*, Lecture Notes in Computer Science **1249** (1997), pp. 252–255.

[8] Coq Development Team, "The Coq Proof Assistant Reference Manual Version 8.0," INRIA-Rocquencourt (2005),
URL: http://coq.inria.fr/doc-eng.html.

[9] Corbineau, P., *Declarative proof language for Coq* (2006),
URL: http://www.cs.ru.nl/~corbineau/mmode.en.html.

[10] Cruz-Filipe, L., H. Geuvers and F. Wiedijk, *C-CoRN, the constructive Coq repository at Nijmegen.*, in: A. Asperti, G. Bancerek and A. Trybulec, editors, *MKM*, Lecture Notes in Computer Science **3119** (2004), pp. 88–103.

[11] Davies, J., "Wiki Brainstorming and Problems with Wiki Based Collaboration," Master's thesis, University of York (2004).

[12] Gottschall, C., *Logic gateway* (2005),
URL: http://logik.phl.univie.ac.at/~chris/gateway/.

[13] Kohlhase, M., *Omdoc: Towards an internet standard for the administration, distribution, and teaching of mathematical knowledge.*, in: J. A. Campbell and E. Roanes-Lozano, editors, *AISC*, Lecture Notes in Computer Science **1930** (2000), pp. 32–52.

[14] Le Hors, A., P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion and S. Byrne, *Document Object Model (DOM) Level 3 Core Specification, Version 1*, W3C Recommendation (2004).

[15] McBride, C., *Epigram: Practical programming with dependent types.*, in: V. Vene and T. Uustalu, editors, *Advanced Functional Programming*, Lecture Notes in Computer Science **3622** (2004), pp. 130–170.

[16] Melis, E., J. Büdenbender, G. Goguadze, P. Libbrecht and C. Ullrich, *Knowledge representation and management in activemath.*, Ann. Math. Artif. Intell. **38** (2003), pp. 47–64.

[17] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL - A Proof Assistant for Higher-Order Logic," Lecture Notes in Computer Science **2283**, Springer, 2002.

[18] Pottier, L., *LogiCoq* (1999),
URL: http://wims.unice.fr/wims/wims.cgi?module=U3/logic/logicoq.

[19] Urban, C., *Implementation of proof search in the imperative programming language pizza.*, in: H. C. M. de Swart, editor, *TABLEAUX*, Lecture Notes in Computer Science **1397** (1998), pp. 313–319.

[20] Web APIs Working Group, *The XMLHttpRequest Object*, Technical report, W3C (2006), URL: http://www.w3.org/TR/XMLHttpRequest/.