Integrating an interactive Haskell tool with a Web Application

an Experience Report

Sylvia Stuurman
Open University, Netherlands
Sylvia.Stuurman@ou.nl

Johan Jeuring
Universiteit Utrecht and Open University, the Netherlands
Johan.Jeuring@ou.nl

Abstract

At the Open University, The Netherlands, we are developing interactive Exercise Assistants that give good feedback to students trying to solve mathematical or logical exercises. To simplify installing, maintaining, and adapting the tools, and to improve reporting facilities, we have turned our tools into web applications. Since our tools are implemented in Haskell, this implies that we have to be able to connect an interactive Haskell application to the web. We have developed an architecture that makes it possible to change an application written in Haskell into a light-weight webservice for an Ajax-style web-based application.

In this paper, we discuss the various possibilities to combine Haskell and a web-based application. We investigate the advantages and disadvantages of the chosen architecture with respect to changes in the interface of the tool written in Haskell.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures; D.2.12 [*Software Engineering*]: Interoperability; H.5.4 [*Hypertext/Hypermedia*]: Hypertext/Hypermedia

General Terms Design, Experimentation

1. Introduction

1.1 Haskell and Web Applications

Techniques to use Haskell in web applications generally aim to use Haskell for all aspects of the application. For example, Meijer et al. [16] introduce Haskell server pages, which treat HTML or XML fragments as ordinary Haskell expressions. HaskellScript [17] has been introduced as an alternative for Javascript. The Haifa project [5] aims ”to build a set of tools, which will enable interoperability applications to be built in the purely functional programming language Haskell using technologies such as SOAP and WSDL”, and WASH/CGI [24] is a set of server-side scripting languages based on Haskell for the generation of XHTML documents and forms, for sessions, and for persistence. Cooper et al. [3] introduce Links, a model-based approach: the presentation layer, the logical layer and the database layer are generated from code written in Links, a functional language based on O’Caml.

These techniques can be used to build a web application from scratch, but they don’t offer a solution to integrate Haskell programs into existing web applications.

1.2 Exercise Assistant in Haskell

At the Open University, The Netherlands, we are developing several exercise-assistant tools: tools in which a student stepwise constructs a solution to an exercise. Examples of exercises the tools support are rewriting a logical expression to disjunctive normal form [13], and solving a system of linear equations [22].

The user-interface of the tools is simple: a student is presented with a text field that contains an exercise in which the student rewrites the exercise toward a solution. After each step, the student can press a Submit button and receive feedback, which appears in the feedback field. The distinctive feature of our tools is the feedback the tools give when a student makes an error. Furthermore, the tool keeps the history of the steps the student performs, and the student can undo previous steps.

To implement our tools, we need functionality for parsing, pretty-printing, symbolic evaluation, several analyses, etc. This functionality builds, traverses or folds abstract syntax trees. Furthermore, the exercise-assistant tools for the different domains (logical expressions, linear equations) are very similar, and we want to minimize code duplication. The lazy higher-order functional programming language Haskell [10] is particularly good at manipulating abstract syntax trees, and the high level of abstraction support by Haskell minimizes code duplication, so we have implemented our tools in Haskell. Furthermore, using generic programming techniques will further reduce code duplication [12]. These techniques are widely available for Haskell [9], and hardly for any other programming language.

1.3 Exercise Assistant on-line

We want to turn our Exercise Assistants into web applications for several reasons. First, the exercise-assistant tools have been developed recently, and are still evolving. Yet we want our students to use the most recent versions of our tools. Deploying an evolving tool is difficult. Deployment in the form of an on-line version of the tool, maintained at a single location, is highly desirable. A web application has the advantage that both the logical part and the presentational part of the application are located at a single location. Therefore, both parts can be maintained without the need of upgrading the application on user machines. Second, the distinguishing feature of our tools is the feedback they give to the student. To improve feedback, we want to log errors and feedback messages. Logging is very hard if not impossible if the tools are installed on user machines. It is much easier to connect a web application to a database and store all errors and feedback messages. Third, we are also considering providing feedback to teachers about common er-

rors made by groups of students. Again, collecting such feedback is much easier in a web application in which such a group of users can login.

The reason that we did not make use of the 'Haskell exclusively' techniques described above, is that we would like to be able to use the Exercise Assistant in existing web applications such as Blackboard.

This paper shows how an interactive Haskell program can be integrated into existing web applications. We describe the requirements for an interactive web application that uses Haskell for its functionality in section 2. Section 3 discusses the various techniques and architectures available for tying an interactive Haskell to the web, and shows our solution, in which the tool is tied in as a light-weight webservice. Section 4 discusses future research around our web application.

2. Requirements

We have the following requirements for an on-line version of our Exercise Assistant.

1. *Interactivity*. The application should have a response behavior resembling a desktop application: there should not be a page reload after each equation or formula that a student submits.
2. *Presentation*. It should be possible to present the web application using different presentation mechanisms. In other words, the functionality of the feedback tool should be separated from the presentation. Then it is, for example, possible to fully integrate the application in a Blackboard course [1] or a Moodle course [19].
3. *Authentication*. For the same reason, authentication should be separated from the Exercise Assistant application, so that authentication from the environment of the user (such as a Blackboard site) may be used.
4. *Scalability*. It should be possible to support the use of the Exercise Assistant by many users at the same time.
5. *Flexibility*. In its present form the tool only covers two domains, does not make use of the history of errors of a particular user, and does not analyse the results of a group of students. In the future we want to adapt our tools at least with respect to these points, but we expect many other changes to be implemented. It should be relatively easy to make changes to the tool, preferably in a single location. The flexibility requirement can be further refined as follows:
   (a) *Transparency*. It should be transparent for the Exercise Assistant whether it resides in a desktop application with a GUI or in a web application. Then the Exercise Assistant can evolve without having to apply changes in different versions of the Exercise Assistant.
   (b) *Stateless connections*. If possible, the web application should adhere to the REST style (Representational State Transfer) [4]. In REST, interactions between client and server are stateless. The REST architecture is the architecture that has made the web as scalable as it is, and adhering to its principles will at least make it possible for a web application to be flexible with respect to changes and scaling.
   (c) *Changeability*. It should be possible to apply changes in the Exercise Assistant without having to apply changes on the various web servers.

3. Toward a solution

3.1 Communication through HTTP

In order to meet requirement 2, we clearly need communication between the Exercise Assistant in Haskell and several webservers. In that case, we would meet requirement 3 at the same time. The communication crosses machine boundaries, so HTTP is an obvious choice as a protocol. The Exercise Assistant as we have developed it is stateless, so requirement 5b is met as well.

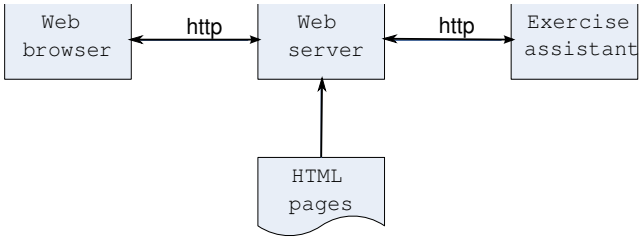

Figure 1. The Exercise assistant on-line

Figure 1 shows how the resources for the web application around the Exercise Assistant are distributed over different machines. The Exercise Assistant may communicate with several web servers, each of which may serve several browsers. Note however, that we still have to find a way to add communication over HTTP to the Exercise Assistant.

3.2 Interactivity through Ajax

The interactivity requirement 1 suggests to use of Ajax [7], which, in short, implies that when a user submits a rewritten expression, the action that follows is not an HTTP-Post request communicating the input to the server and resulting in a page reload, but an XMLHTTP Post request, communicating the input and receiving the feedback without a page reload. Code in a scripting language in the page (in practice, the scripting language is almost always Javascript) performs the action, and shows the response in one or more components of the page. Because the web browser does not need to render a whole page, Ajax-based applications have a response time almost resembling a desktop application. By using the Really Simple History framework [20], it is possible for the user to undo previously submitted rewritings.

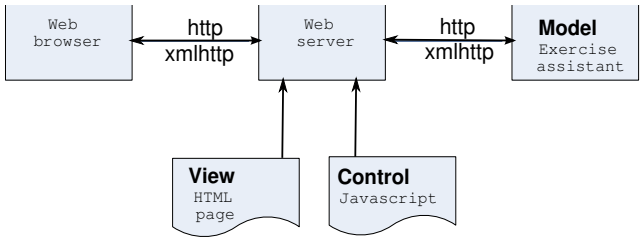

Figure 2. The Exercise assistant on-line with Ajax

Figure 2 shows that the webserver not only stores HTML pages, but also Javascript code which is needed for the Ajax communication with the Exercise Assistant. The resources of the web server will be sent to the browser, where the HTML is displayed, and the Javascript code is interpreted. In fact, this architecture is an example of the MVC architecture, with the Exercise Assistant as the model, the HTML pages as the view, and the Javascript code as the controller (with the possibility to apply changes in the HTML page while it is displayed in the browser).

When comparing the properties of this architecture with the requirements, we may conclude that requirement 1 is met by using Ajax. Requirement 2 is met by allowing different web servers to communicate with the Exercise Assistant, and keeping the responsibility of the look and feel of the application with the webserver. The same properties also guarantee requirement 3. Requirement 4 is met because the work load can be distributed over several web servers. However, there will be a limit to the number of web servers that the Exercise Assistant can serve. Requirement 5b is met by the nature of the Exercise Assistant, because it functions without state and without side-effects.

Requirement 5 however, is not met entirely. In the first place, we still have to find a way to establish communication through HTTP between the Exercise Assistant and the web server, so we still do not know whether such a mechanism guarantees requirement 5a. And requirement 5c is only met for changes in the implementation of the Exercise Assistant; not for changes within the interface for the XMLHTTP requests.

3.3 Application server

To implement the architecture sketched in 2 we need a mechanism for communicating between a program written in Haskell and a program that is able to communicate over HTTP: a web server. A number of techniques for calling a Haskell function from a webserver are available.

Program Call. When the webserver supports server-side scripts such as PHP, JSP or ASP, a script can simply call an executable of a Haskell program with the input of the user as a parameter, and send back the result to the user. The disadvantage of such a solution is that each time a user presses a submit button, a new process is started with an associated time delay, violating the 'interactivity' requirement 1. Moreover, with many users working at the same time, the number of processes may become a bottleneck, violating the 'scalability' requirement 4.

CGI. CGI, the Common Gateway Interface, is a standard for programs to communicate with web servers. With CGI, it is possible to use a program without a scripting language like PHP. However, using CGI has the same problems as the previous technique: when a user presses submit, a new process is started.

Server-side scripting. Haskell Server Pages [18] treat HTML or XML fragments as ordinary expressions. It is possible to refactor our Exercise Assistant to Haskell Server Pages and thus turn it into a web-service. However, we would have to maintain both a desktop version of the tool and an on-line version, violating the 'transparency' requirement 5a.

FastCGI. FastCGI [14] is a fast web server interface that solves the performance problems inherent in CGI. It uses a persistent process instead of a process for each request, like CGI. There is a FastCGI implementation for Haskell [2]. To make use of FastCGI, we would have to write a program which is capable of scheduling the Exercise Assistant in several threads, and keeping track of sessions if needed.

Apache module. In the same way as the Apache web server supports scripting languages like PHP, we can use an Apache module supporting the interpretation of Haskell source code. Such a module is available [8]. There are issues that have to be solved (the Haskell interpreter is not thread-safe for instance) before this would be a viable option. Another disadvantage is that interpretation of the Exercise Assistant might be too slow with respect to the 'interactivity' requirement 1.

Application server. A start for an application server for web services could be Marlow's web server in Haskell [15]. Another candidate is the HTTP server of the HAIFA project [5, 6], which offers a simple HTTP server with handlers to be built in as Haskell functions. A third possibility is the HAppS application server, which is the most complete server at this moment [11], supporting sessions and DBMS access without having to use a monad. We choose this application server for our web application.

By using the HAppS server, we have met the 5a requirement, because the HAppS server calls the same functions of the Exercise Assistant as the user interface of the desktop version of the Exercise Assistant.

3.4 Problems

However, the 5c requirement is not yet met: the interface of the HAppS server will change when the functionality of the Exercise Assistant changes. The HAppS server is configured by writing a main module. Figure 3 shows part of the main module. There

```
app POST _ ["logic", "feedback"] = do
     rq <- getEvent
     sresult 200 (giveLogicFeedback
          (lookS 200 rq "answer")
          (lookS 200 rq "previousanswer"))
```

Figure 3. part of the Main Module of the application server

is a clause that declares what will be sent back over HTTP, in case of an incoming (xml-)HTTP Post request. If the URL of the request contains the strings `logic` and `feedback`, the values of the variables `answer` and `previousanswer` sent with the request are used as parameters of the function call `giveLogicFeedback` of the Exercise Assistant.

This approach has two disadvantages:

Compilation

The only way to add or change a clause coupling a pattern in the request to a function call is by adding the clause to the main module of the HAppS server, and compile the whole server, with the Exercise Assistant included. Even if we change the Exercise Assistant internally, without changing the interface, the whole server has to be recompiled. In a production server that is unacceptable.

A possible solution would be to have a configuration file that couples patterns in the URL of a request to the name of the function to call, combined with pluggable functions. One of the patterns could have an associated action to reread the configuration file. In that case, functions can be compiled separately from the application server, and there would be no downtime. Using the existing plugin-technology for Haskell [21], such a solution is possible, at least in theory. However, the plug-in technology is not yet stable enough to be able to rely on it.

Another option is to choose the FastCGI solution instead of the application server. The disadvantage of such a solution is that we would have to write code for functionality that is already available in the HAppS application server, such as scheduling of threads, working with sessions, and database access.

For now we stick to the application server solution, but we may have to choose another technique if we have to apply changes very often.

Tightly coupled Model, View, and Control

Changes in the interface have consequences for the resources on the web server. The requests to the application server contain variables, to be used as input for the Exercise Assistant. The names of these variables should be known by the Javascript code that is responsible for the Ajax communication between the browser and

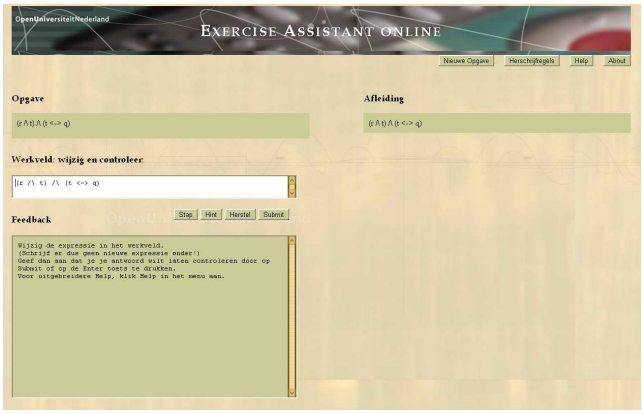

Figure 4. The Exercise assistant

the application server. The Javascript code needs to know which page components hold the values to send, which variable, and in which page components to put the results. In figure 4, the variable named `answer` should get the value of the user input, while the Javascript code should remember the value for the variable named `previousanswer`. The feedback that is sent back to the browser should be pasted in the page component in the lower left of figure 4. Obviously, both the HTML and the Javascript code should be changed when the interface of the Exercise Assistant needs more values in the request, or sends more values back.

When the coupling between a presentation layer and a logic layer is too tight to keep changes in one layer, the solution is often a model-based approach. A web application is modeled as a single entity, and from the models, the presentation layer is generated. An example of that approach is OOWS [23]. A model-based approach might even be used to generate the presentation layer, the logical layer and the database layer as in Links [3]. In our architecture, that is not a viable option, because we want to keep the possibility to use the Exercise Assistant from different web servers, which will, in practice, not be under our control.

3.5 Model, View and Control in the Application Server

A solution to the problems of the previous subsection would be to keep the HTML and Javascript on the application server, but that would violate requirements 2 and 3. The solution which meets all requirements therefore, is to store the Javascript code in the application server, and shift the responsibility for the components that are needed of the HTML page to the Javascript code. A web developer who wants to use the Exercise Assistant then uses the URL for the Javascript code in a web page, and calls a Javascript function which will add the necessary components in the page. Figure 5 shows the

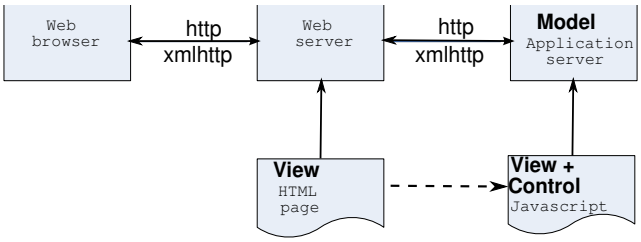

Figure 5. The Exercise assistant web application

distribution of resources in this solution. The responsibility for the Control is shifted to the application server. Because of the tight coupling mentioned above, the Control component provides the page components that are vital for the application as well. They will be pasted in the web page that is provided by the web server. The dotted arrow from the View on the Web server to the View and Control on the application server shows that the web page (served by the web server) contains a reference to the Javascript code (stored on the application server).

3.6 The Exercise Assistant on-line

The on-line Exercise Assistant in figure 4 is started when the user types in the URL in the browser (at the moment the Exercise Assistant can be found on `http://www.exercise-assistants.org/feedback/logic/`. The domain-name will remain stable, the postfix feedback might change in the future.) A simple HTTP Get request is sent to the web server. The web server (in our reference implementation an Apache web server supporting PHP) sends a page to the browser, containing Javascript code from the application server. The Javascript code, when interpreted in the browser, asks the application server to generate an exercise using an XMLHTTP request, and shows the resulting exercise in an editable element that it has added to on the page. The user then starts to solve the exercise. When the user presses the submit button, the necessary data are sent to the application server in an XMLHTTP request by the Javascript code in the page. The application server, after having called the appropriate function of the Exercise Assistant, responds by sending feedback to the browser, where Javascript code pastes the result in the right places in the page. Figure 6 shows the Exer-

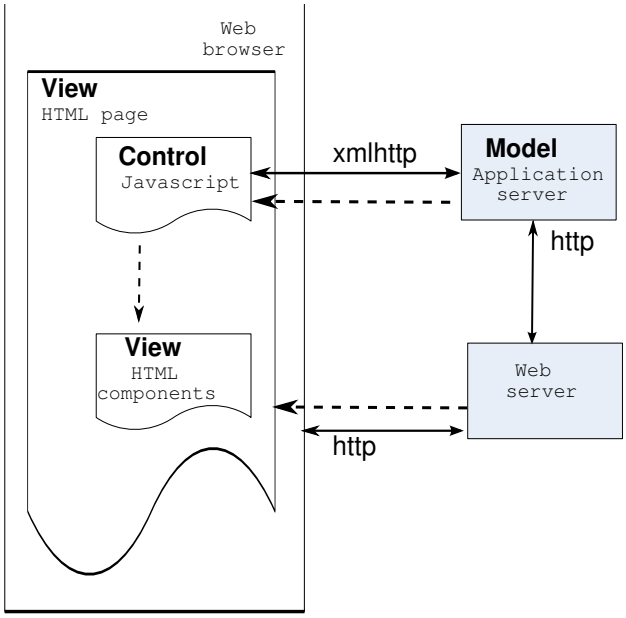

Figure 6. The Exercise assistant web application

cise Assistant application at run-time. The browser has received an HTML page from the webserver (as indicated by the dotted arrow), which contains Javascript code that has been served by the application server (as indicated by the dotted arrow). The Javascript pastes the necessary page components in the HTML page (as indicated by the dotted arrow), and from then on sends the input from the user to the application server, and pastes the answers in the right page components.

More than one webserver may connect to the application server, and there are no restrictions regarding the technologies used in the web server.

4. Conclusions and future work

We have shown that we can use existing techniques to turn an interactive Haskell application into a web-based application. We have listed a number of requirements that a solution should satisfy, and we have discussed if and how several alternative approaches available for Haskell applications satisfy these requirements. We hope to contribute to the discussions about architectures for Haskell web applications[1].

Generation of Javascript code

A web application like the one we present here is hard to debug because a bug may be located in different places, and the application is built using several languages. Changing the Javascript code by hand each time there is a change in the interface of the Exercise Assistant will probably result in extra hours spent searching for bugs. A possible solution might be to evolve a language like Links [3], but because our Exercise Assistant is developed as a desktop application, it is easier for us to develop and test the functionality of the Exercise Assistant stand-alone, and find a way to generate the Javascript-code that we need.

Valid Selections

Another future direction in our experiment is to enhance the editing capabilities of the application and to introduce the concept of strategies. For each domain, we would like to be able to express different strategies that a user may follow while solving an exercise. Strategies could be used to offer the user a set of possible transformations for a given selection in the exercise at hand, or to give feedback about the strategy that is used.

With respect to responsiveness, it will be necessary to build the capability to recognise which selections in an expression are valid or not, into the Javascript component. For example, in an expression $a * (b + c)$, $b + c$ is a valid selection, while $a * (b$ is not.

We are planning to extend the Javascript code with this capability. The Control component will then be able to tell the application server which part of an expression is selected. We will need ways to specify the rules for valid selections in different domains, and these specifications will be used both by the Exercise Assistant in Haskell and the Control component in Javascript.

So, switching the responsibility for the Javascript code to the application server is not only needed for the flexibility with respect to changes in the Exercise Assistant, it is also needed to turn the browser into a smart editor that knows which selections in an expression are valid.

Acknowledgements. Harrie Passier and Josje Lodder helped in the implementation of the Exercise Assistant, and commented on several aspects of the web application.

References

[1] The blackboard learning system. `http://www.blackboard.com/`.

[2] B. Bringert. fastcgi - a Haskell library for writing FastCGI programs. `http://www.cs.chalmers.se/~bringert/darcs/haskell-fastcgi/doc/`, 2006.

[3] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. `http://groups.inf.ed.ac.uk/links/papers/links-fmco06.pdf`, 2006.

[1] See for instance the fa.haskell Google group.

[4] R. T. Fielding and R. N. Taylor. Principled designn of modern web architecture. In *Proceedings of 22nd International Conference on Software Engineering*, pages 407–416, June 2000.

[5] S. Foster. HAIFA: An XML based interoperability solution for Haskell. In *Proceedings of the 6th Symposium on Trends in Functional Programming (TFP 2005)*, pages 103–118, 2005.

[6] S. Foster. HAIFA, 2006. `http://www.dcs.shef.ac.uk/~simonf/HAIFA.html`, accessed june 2006.

[7] J. Garrett. Ajax: A new approach to web applications, 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[8] A. Hemel and E. Dolstra. Mod Haskell. `http://losser.st-lab.cs.uu.nl/mod_haskell/docs/mod_haskell/manual`.

[9] R. Hinze, J. Jeuring, and A. Löh. Comparing approches to generic programming in Haskell. In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Generic Programming, Advanced Lectures*, LNCS. Springer-Verlag, 2007.

[10] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, 2007.

[11] A. Jacobson. HAppS – haskell application server. `http://happs.org/`, 2006. `http://happs.org/`, accessed November 2006.

[12] J. Jeuring, H. passier, and S. Stuurman. A generic framework for developing exercise assistants. In *The 8th International Conference on Information Technology Based Higher Education and Training, ITHET'07*, 2007.

[13] J. Lodder, J. Jeuring, and H. Passier. An interactive tool for manipulating logical formulae. In M. Manzano, B. Pérez Lancho, and A. Gil, editors, *Proceedings of the Second International Congress on Tools for Teaching Logic*, 2006.

[14] O. Market. Fast CGI Whitepaper. `http://fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm`, 1996.

[15] S. Marlow. Developing a high-performance web server in Concurrent Haskell. *Journal of Functional Programming*, 12(4, 5):359–374, July 2002.

[16] E. Meijer. Server side web scripting in haskell. *Journal of Functional Programming*, 10(1):1–18, January 2000.

[17] E. Meijer, D. Leijen, and J. Hook. Client-side web scripting with HaskellScript. In *Proceedings PADL'99*, volume 1551 of *Lecture Notes in Computer Science*, pages 196–210. Springer-Berlag, 1999.

[18] E. Meijer and D. v. Velzen. Haskell server pages, functional programming and the battle for the middle tier. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001.

[19] Moodle, a free, open source course management system for online learning. `http://moodle.org/`.

[20] B. Neuberg. Ajax: How to handle bookmarks and back buttons, 2005. `http://www.onjava.com/pub/a/onjava/2005/10/26/ajax-handling-bookmarks-and-back-button.html`.

[21] A. Pang, D. Stewart, S. Seefried, and M. M. T. Chakravarty. Plugging Haskell in. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, 2004.

[22] H. Passier and J. Jeuring. Feedback in an interactive equation solver. In M. Seppälä, S. Xambo, and O. Caprotti, editors, *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 53–68. Oy WebALT Inc., 2006.

[23] O. Pastor, J. Fons, and V. Pelechano. OOWS: A method to develop web applications from web-oriented conceptual models. In *International Workshop on Web Oriented Software Technology (IWWOST)*, 2003.

[24] P. Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Transactions on Internet Technology*, 5(1):1–46, 2005.