Turning an interactive tool implemented in Haskell into a web application – An experience report

Sylvia Stuurman

Johan Jeuring

Department of Information and Computing Sciences, Utrecht University

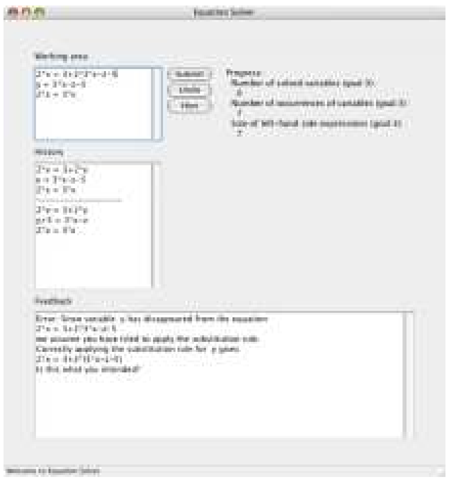
Technical Report UU-CS-2007-008

www.cs.uu.nl
ISSN: 0924-3275

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.



Number of solved variables (goal 3):
0
Number of occurrences of variables (goal 3):
7
Size of left-hand side expression (goal 3):
7

Error: Since variable y has disappeared from the equation
2*x = 3+2*3*x-2-5
we assume you have tried to apply the substitution rule.
Correctly applying the substitution rule for y gives
2*x = 3+2*(3*x-2-5)
Is this what you intended?

Figure 1: The equation solver

1 Introduction

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 [14], and solving a system of linear equations [20]. Figure 1 shows a screenshot of our desktop application that supports solving a system of linear equations.

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 towards 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 [12] 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.

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 errors 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.

This paper investigates the various possibilities for tying an interactive Haskell program to the web. 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 as a light-weight webservice into a web-based application. Section 4 investigates the problems we encounter with such a solution if we want to change the tool, and discusses possible directions to solve those obstacles. The solution for the integration of a Haskell program in a web application is, obviously, specific for tools written in Haskell; the problems caused by changes in a system based on light-weight web services are problems that manifest themselves independently of the programming language behind a web service.

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 that resembles that of 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, a Moodle course, or an ASP application.

3. Authentication. For the same reason, authentication should be separated from the feedback 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), documented in [6]. In REST, interactions between client and server are stateless. A client request for information is performed through an HTTP Get request; a client providing information that may change the information on the server is performed through an HTTP Post request.
   The REST architecture is the architecture that has made the web as scalable as it has shown to be, and adhering to its principles will at least make it possible for a web application to be flexible with respect to changes and scaling.

Most of these requirements are standard requirements for interactive web-based applications.

3 Solution

Figure 2 shows how the on-line version of the exercise assistant is distributed over different machines, with one application server communicating with several web servers, each of which communicates with several browsers. According to the 'presentation' (2) and 'authentication' requirements (3), the exercise assistant should be able to communicate with several different web servers

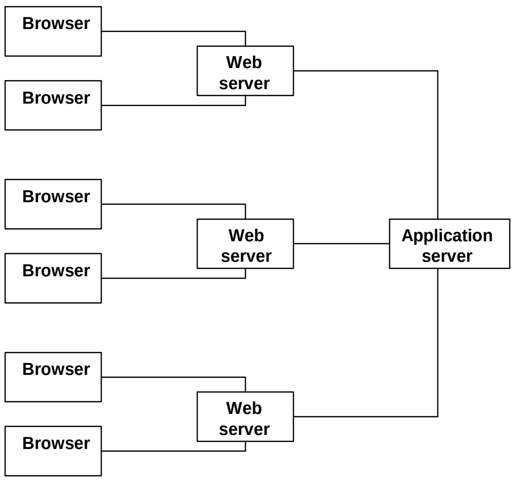


Figure 2: Deployment architecture

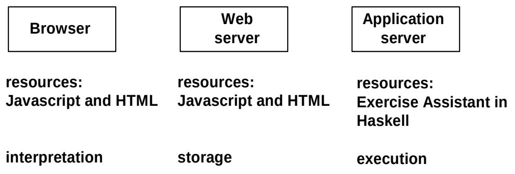

Browser
Web server
Application server
resources: Javascript and HTML
resources: Javascript and HTML
resources: Exercise Assistant in Haskell
interpretation
storage
execution

Figure 3: Resources

(for instance hosting a Blackboard environment), each of which serves several clients in the form of browsers.

The 'interactivity' requirement (1) suggests to make use of Ajax [10], 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 request, communicating the input and receiving the feedback without a page reload. Code in a scripting language in the page (in practice, that scripting language is almost always Javascript) performs that action, and shows the response in one or more elements of the page. Because the web browser does not need to render a whole page, Ajax-based applications have a response time that almost resembles that of a desktop application.

By using the Really Simple History framework [18], it is possible for the user to undo previously submitted rewritings.

The application server in figure 2 receives input from different web servers, and responds by sending the feedback. In the architecture we propose, the application server functions as a light-weight web service, without the overhead of SOAP and related standards, relying on simple HTTP and XMLHTTP requests. To implement the application server, we need a web server that can communicate with the exercise assistant.

Figure 3 shows how the resources for the on-line Exercise Assistant are distributed over the browser, the web server and the application server. The web server stores the HTML and Javascript, the browser interprets the HTML and Javascript, and the Exercise Assistant is executed in the application server.

3.1 Techniques for calling Haskell applications from a webserver

There exist a number of techniques for calling a Haskell function from a webserver.

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. In [17], Haskell Server Pages are proposed, which 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.

COM objects. In [7], a technique is described to package a Haskell program as a COM object. Such a COM object may be called from an ASP page, and there are solutions for other scripting languages as well. The constraint of such a solution is that COM objects need a Microsoft platform. And this solution has the same problem as the first two solutions: for every request, a new process is started.

FastCGI. FastCGI [15] 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 [3]. 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.

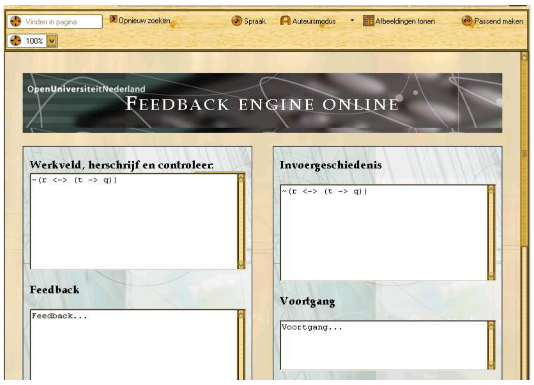
Apache module. In the same way as the Apache web server supports scripting languages like PHP, we can use an Apache module supporting the interpretation of Haskell source code. Such a module is available [11]. There are issues that have to be solved (the Haskell interpreter is not thread-safe for instance) before this could 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 the web server in Haskell described in [16]. Another candidate is the HTTP server of the HAIFA project [8] and [9], 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 [13], supporting sessions and DBMS access without having to use a monad. It is this application server that we chose for our web application.

3.2 The on-line Exercise Assistant

The on-line Exercise Assistant in figure 4, is started when the user types in the URL in the browser[1]. 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 containing Javascript code to the browser. 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 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 exercise assistant, responds by sending feedback to the browser, where Javascript code pastes the result in the right places in the page, the Ajax-way.

[1]At the moment the Exercise Assistant can be found on `http://www.exercise-assistants.org/feedback/`. The domain-name will remain stable, the postfix `feedback` might change in the future.



Werkveld, herschrijf en controleer:

```
~(r <-> (t -> q))
```

Invoergeschiedenis

```
~(r <-> (t -> q))
```

Feedback

```
Feedback...
```

Voortgang

```
Voortgang...
```

Figure 4: On-line Exercise Assistant

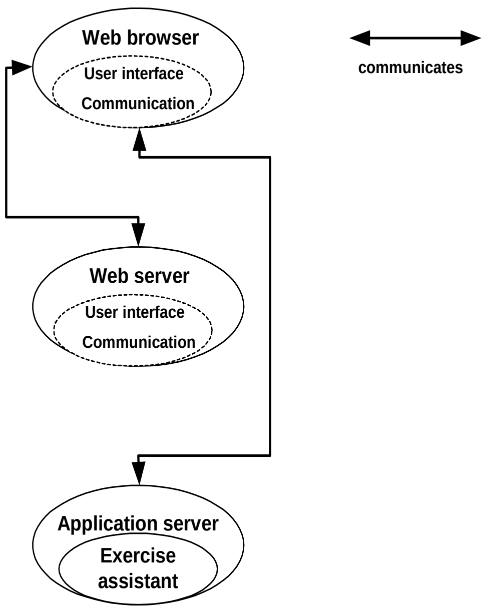


Figure 5: Components

Figure 5 shows the components of the architecture at run-time. The arrow between the Browser component and the Web server component means that the Browser component requests a page from the Web server component, and the Web server component sends a page (the user interface) including Javascript code, which will function as the Communication component. Both the user interface and the Communication component are stored in the Web server component, and are interpreted in the Browser component. The Communication component communicates with the Application server component, which calls the Exercise Assistant to receive feedback.

We have implemented the system using Apache as a webserver with support for PHP, and HAppS as a Haskell application server. More than one webserver may connect to the application server, and there are no restrictions regarding the technologies used in the web server.

4 Problems and possible remedies

Although the architecture has been chosen with flexibility in mind, there are some problems with respect to changing the web application. These problems are caused by the Haskell application server, and by the decision to implement the web application as a light-weight webservice.

4.1 Haskell application server-specific problems

The Haskell application server checks incoming requests and calls the appropriate function of the feedback engine. The application server code is compiled together with the feedback engine code to achieve this functionality. That means that for each change, the server (including the exercise assistant) has to be shut down, recompiled, and restarted.

Remedy A solution for this Haskell application server-specific obstacle 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 [19], 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 Haskell 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.

4.2 Light-weight webservices related obstacles

Other problems for changing the web application are caused by the implementation of the exercise assistant itself.

- At the moment, **exercise assistant** expects two strings as input: one containing the expression that a user has entered, and one containing the expression from the previous correct attempt (or the generated exercise with which the user started). This means that the **application server** has to call the exercise assistant with two strings as input.
- The **application server** expects a request with a value for a variable named "answer" and a value for a variable named "previousanswer". These values (both strings) are input for the exercise assistant. This means that the **communication component** (in Javascript) should send an XMLHTTP-Post request with a value for a variable named "answer" and a value for a variable named "previousanswer". It also means that the **web page** should allow the user to type in an answer, and that the **communication component** should remember

the previous answer and know from which element of the page it can extract the value for "answer".

- The **exercise assistant** returns two strings and a boolean when it has been called. This means that the **application server** should use those two strings and the boolean to compose a return message.
- The **application server** returns a value for a variable named "feedback", a value for a variable named "progress", and a value for a variable named "consistent". The **communication component** should know what to do with these values, and it should know which elements of the page should be updated.

It follows that any change in the feedback functionality of the exercise assistant that involves a change in input or response has consequences for the application server, for the web page, and for the communication component in the web page. The same applies for the generation of exercises.

Some examples of such changes are:

- We might want to offer the possibility to give feedback in different languages (English, Dutch, Spanish,...). Then the start page of the exercise assistant would offer the user a choice of languages, and either the communication component would send the preferred language with each request, or the preferences per user would be saved in the application server, and the user would be known by name. In both cases a request for the application server contains an extra parameter.
- When the exercise assistant returns three instead of two strings (for instance showing the progress of the user at each step), the application server will have to encode that third string into the response to the browser as well. We will have to update both the HTML page (add another text field) and the communication component (read another field of the response, and show it in the added text field).
- Similar measures have to be taken when we want to store the steps of individual users, analyse those steps as input for the feedback engine, or add user authentication.

It is obvious that we run into problems with the above changes. We expect not only to change the implementation of our light-weight web service, but also its interface. With a change of the interface, changes are needed in the application server, in the web page, as well as in the communication component.

Remedy When the coupling between a presentation layer and a logic layer is too tight, the solution is often sought in 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 [21]. A model-based approach might even be used to generate the presentation layer, the logical layer and the database layer [5, 1].
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 practise, not be under our control.

In the area of webservices, the solution for this problem is to describe capabilities and properties of web services in a specific language that should be understood by the requester of a web service. DAML-S [2] is a language for specification of the semantics of a web service; WSDL [4] is a language for a communication-level description of the messages and protocols. The problem of the alignment of the user interface with a changing interface of web services, or a changing logical model, has been observed often. In [22] for instance, the authors mention the problem of "synchronization of the human user interface with the BPEL engine", in webservices that are "choreographed" as a conversational process (BPEL is a language for such choreography). Even the sequence of the webservices used as such is difficult to synchronize with in the user interface: "If the page flow-based approach is chosen, the synchronization of the user interface and the BPEL engine is

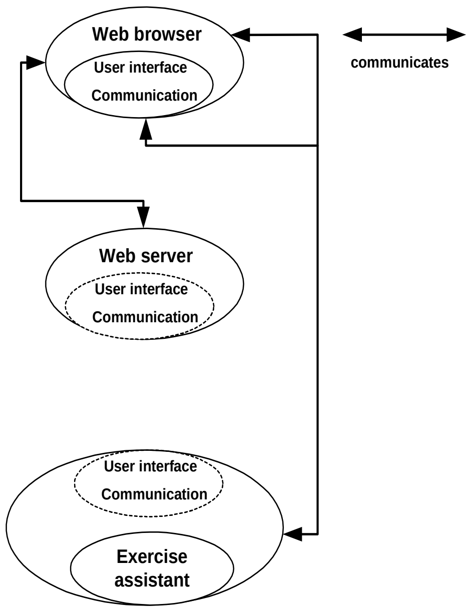


Figure 6: Components

a major challenge, introducing mutual dependencies between the user interface and the BPEL process. These dependencies must be designed explicitly."

In our case, we opt for a rather simple solution. In the architecture that we have described, depicted in figure 5, the communication component (in Javascript) is the responsibility of the web server: it is sent from the web server to the browser. The reason is that the elements of the web page are tightly coupled to the Javascript code of the communication component, because the communication component should know from which elements to retrieve the data, and in which elements to display the response.

In figure 6, we show the solution. The responsibility for the communication component is shifted to the application server. Because of the tight coupling mentioned above, the application server should provide the elements of the page that are vital for the application as well. They will be pasted in the web page that is provided by the web server.

The web server thus sends a page to the browser, while the application server sends the communication component, and those page elements that are needed by the communication component. The page itself, and the stylesheet that determines the look of the elements is determined by the web server; the Javascript and the page elements it needs, are determined by the application server.

4.3 Debugging

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.

Remedy It is highly desirable to find a way to specify what we need of the Javascript code, and generate the code and the page elements each time that there is a change in the specification.

5 Conclusions and future work

We have shown that we can use existing techniques to turn an interactive Haskell application into a web-based application. Furthermore, we have investigated the problems caused by the many changes that we foresee in the Haskell application, and discussed solutions to those problems.

Some of those problems are caused by the fact that our application is written in Haskell, using a Haskell application server, and some are the result of our wish not only to be able to change the implementation of the webservice, but also the interface.

In the future, we will implement the remedies we have formulated in this paper, to see if we are able to implement a web-based application with the inherent property that changes in the web service are possible without the need to apply changes elsewhere.

Strategies and Editor 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 probably be necessary to build the capability to recognise wich selections in an expression are valid and which are not valid, into the Javascript component. In an expression $a * (b + c)$ for instance, $b + c$ is a valid selection, while $a(b$ is not.

We will extend the Javascript code with this capability. The communication component will then be able to communicate to the application which part of an expression is selected. We will need ways to specify the rules for viable selections in different domains, and those specifications will be used both by the Exercise Assistant in Haskell and the communication component in Javascript.

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

References

[1] P. Achten, M. v. Eekelen, and R. Plasmeijer. Generic Graphical User Interfaces. In *The 15th International Workshop on the Implementation of Functional Languages, IFL 2003, Selected Papers*, volume 3145 of *LNCS*, pages 152–167. Springer, 2004.

[2] A. Ankolekar, B. Burstein, J. Hobbs, O. Lassila, D. Martin, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web service description for the semantic web. In *Proceedings of the 1st International Semantic Web Conference (ISWC)*, 2002.

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

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, W3C Note. Technical report, W3C, 2001.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. `http://groups.inf.ed.ac.uk/links/papers/links-icfp06/links-icfp06.pdf`, 2006.

[6] 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.

[7] S. Finne, D. Leijen, E. Meijer, and S. L. P. Jones. Calling hell from heaven and heaven from hell. In *International Conference on Functional Programming*, pages 114–125, 1999.

[8] 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.

[9] S. Foster. HAIFA, 2006. [on-line, accessed june 2006].

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

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

[12] 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.

[13] A. Jacobson. HAppS – haskell application server. http://happs.org/, 2006. [[on-line, accessed NOVEMBER 2006].

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] O. Zimmermann, V. Doubrovski, J. Grundler, and K. Hogg. Service-oriented architecture and business process choreography in an order management scenario: rationale, concepts, lessons learned. In *20th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 301, 312, 2005.