Web Continuations and a C++ Implementation

Sancho McCann
13899067
CPSC 511 Term Paper
Email: sanchom@cs.ubc.ca

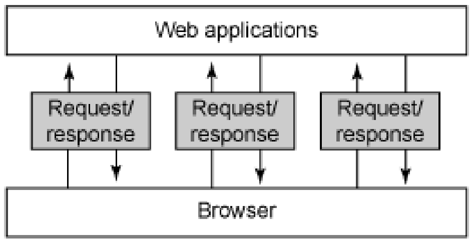

Fig. 1. Typical interaction between a server and a client [15].

Abstract— **The study of web continuations is now at maturity. Their advantage lies in the ease with which state can be maintained despite a stateless internet protocol, overcoming problems for both the developer and the user of a web application. There are various approaches to the continuation's implemenation, subtly differing in capabilities due to design decisions. This paper explores the space of existing implementations and presents a new C++ implementation for comparison and exposition of these decisions.**

I. Introduction

The model of web client-server interaction has undergone a shift throughout the life of the internet. The shift has been one from static, non-interactive web pages to dynamic, interactive web sites. Although this model of interaction has evolved, evolution of supporting programming language abstractions has predictably lagged behind. Only recently have efforts been made to support in languages more directly through continuations a programming model for statefull client-server interaction across the web [4], [7], [13], [14].

Historically, interaction between a web browser and a server was as a series of stateless requests and responses. Figure 1 shows a typical interaction model between a server and a client.

The stateless web was actually an improvement from earlier distributed applications. The server being freed from the requirement to dedicate resources to a specific client throughout its entire interaction session greatly improved performance and scalability [15].

However, there are two main trade-offs that come with this performance gain. First, whenever state is desired, the burden of management is now on the developer. Second, the relationship between client and server has become very asymmetric with the browser clearly in control [13]. The first trade-off affects the programmers; the second trade-off affects users.

```
/* start page 1 */
output form 1
wait for input
process input
go to page 2
/* end page 1 */

/* start page 2 */
check for form 1 data
if not present go to page 1
output form 2
wait for input
process input
go to page 3
/* end page 2 */

/* start page 3 */
check for form 2 data
if not present go to page 2
output form 3
wait for input
process input
go to next page
/* end page 3 */
```

Fig. 2. Example of the type of management required to guide a user through an interactive web application [14].

Today, it is not hard to find examples of web applications in which state is important. Reservation services, online shopping sites, and educational quizzes all fall into this category.

For the developer guiding and tracking a user's path through such an application, implementation of the state-management is a major aspect of the development. The case is that there is a coroutine relationship between the user and the script. To allow this relationship using a stateless protocol, web programmers have typically split the script into fragments and direct the user from fragment to fragment (see Figure 2). Execution stops after each fragment completes - information needed by each successive fragment must be passed via the intermediate user interaction using hidden fields or cookies [11].

This non-direct style of implementation for dynamic content generators does not expose cleanly the expected path through the program. Simple programs that could be written and read sequentially when they targeted a console as a user interface become bloated and non-sequential when targeting a remote web browser as a user interface. Again, the underlying reason for this is the web's statelessness.

This model does not only pose problems for development, but also for a user navigating through such a site. All goes well they follow the path envisioned and prescribed by the developer. Clone the browser window, or use the back-button at the wrong time however, and the resulting behaviour can be very surprising.

Suppose I am shopping at an online store and add a CD to my shopping cart. I then am presented with a page suggesting similar items, so I clone the window and proceed to browse the suggestions. In this cloned window, I add a DVD by the same artist to my shopping cart, only to second-guess this decision and decide to purchase only the CD. So, I close the cloned window I have been browsing in and proceed to check-out in the original window. If I am not careful, I will end up purchasing not only the CD, but also the DVD which has been added to the cart from the cloned window. What was the problem? The server did not view this interaction as a fork in the path - only as sequence of requests: first, my addition of a CD to the cart, second, my addition of a DVD to the cart, and third, my proceeding to check-out.

These problems of non-direct style of implementation and the inability of a user to fork paths of navigation are both solved by the *continuation*.

The first half of this paper discusses the viability of the continuation as a solution to these problems, examines several implementations of the continuation, and addresses criticisms of the continuation as a useful tool and abstraction. The second half of this paper describes an new implementation of a continuation aware web server in C++.

II. The Continuation

The fragmentation of a script as in Figure 2 does come one step closer to modeling the actual user-program relationship, but not exactly. What is missing is a method of dealing with the user's implicit expectation of a multiply resumable co-routine in their use of the back-button and window cloning [11].

A suggested solution to this problem of representation is the continuation: a representation of the program's store and control state. Applied to this problem, a continuation is captured at the time that the program delivers a request to the user for input (more on this use of terminology later). When the user responds, the server is able to trigger the program's continuation and execution continues. Rather than execution terminating with each interaction, execution now simply suspends. This allows "direct-style" programming and supports users' expectations of window cloning and back-button use.

As a contrast to the program consisting of fragmented scripts as in Figure 2, Figure 4 shows a corresponding direct-style implementation. The key to this transformation is the availability of a construct that allows the *wait for input* action by the program; this is the continuation. Continuations allow the use of one piece of code to model the control flow over several web pages [5]. Figure 3 shows a diagram contrasting Figure 1.

As mentioned earlier, this representation also reverses the roles of the user and the program. Without such a representation, the program is purely responsive, triggered into action by users' requests. Now, it is the program delivering the requests and waiting for a response from the user.

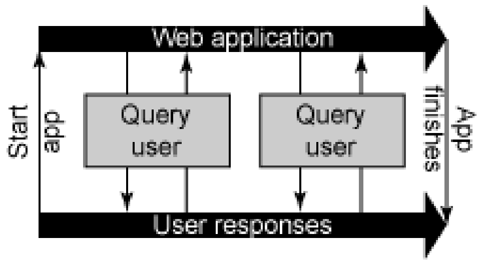

Fig. 3. Interaction with a continuation web server [15].

```
/* start page */
output form 1
wait for input
process input
output form 2
wait for input
process input
output form 3
wait for input
process input
go to next page
/* end page */
```

Fig. 4. Example of the fragmented code from Figure 2 in direct-style [14].

How available are continuations? Continuation mechanisms are available in several languages including Ruby, Scheme, and Smalltalk. Even C's setcontext/getcontext/setjmp/longjmp faculties can be seen as continuation storage and invocation. This might seem like a small list, however, regardless of native language support for continuations, Matthew's et al. [11] describe an automatic program transformation that can be used to change any program in a language supporting a GOTO mechanism into a web-ready program with continuation support.

III. Design Decisions

There are several design decision that stem from one main question when designing a continuation web server: how is the continuation to be represented and stored?

One option is to store the continuation on the server and pass a reference to the continuation to the user as a hidden field in the form they will be completing and submitting. This is the technique used by Graunke et al. [7] and Krishnamurthi [10].

Now that the user holds a reference to the continuation stored on the server, garbage collection of unused continuations becomes an issue. One would need a distributed garbage collection mechanism. Generally, garbage collection works by reference counting: when there are no remaining references to an object, it can be collected. The trouble with distributing references to users is that the server has no way to determine when these references are deleted (when the user closes the browser, or when the user deletes the bookmark).

A potential solution to this problem proposed by Matthews et al. [11] is to treat the distributed references as *weak references* - references to objects that may be reclaimed early but that provide reasonable default results when the subsequent invalid de-reference is attempted. This would be accomplished by removing the continuation from the server after some timeout has expired and to direct the user back to the start of interaction if a restart of a reclaimed continuation is attempted. Matthews et al. further mention that deciding on this timeout value is a very real problem. Too short a timeout and users' interactions will often be restarted; too long a timeout and servers will become overloaded. As discussed next, they reject the idea of server-side storage of continuations altogether and provide no insight into this issue. It seems that in the case that a timeout must be chosen, this should be left to the developer of the individual scripts, with a maximum imposed by the server. This would allow aggressive collection of non-critical continuations, while giving some freedom to the developer to push this back.

What is the alternative to the server-stored continuation? There is only one: give the continuation to the user. Both Queinnec [13] and Matthews et al [11] advocate this solution and discuss how to overcome some of the problems associated with it.

Serializing the continuation into cookies or hidden fields and delivering them to the user has the advantage that there is no need to guess a timeout value. The continuation is never invalid, even after server outages. One issue is that unserializable values must be re-built or re-obtained (files or other system resources) [13]. Queinnec, however, does not mention the fact when a user attempts to resume a continuation, the required system resources may no longer be available, either temporarily or permanently.

One other design decision is whether to use a language that provides continuations natively or to simulate/implement them and provide an abstracted interface to the programmer. Native language support (or automatic conversion as described by Matthews et al. [11]) would allow much closer control of the behaviour of the continuations for specific user-program interaction sequences as compared to an abstracted interface. For example, there are situations when it would be desirable to change the back-button behaviour (as in an educational quiz). It would be difficult to deliver an abstraction that could easily handle many special cases such as this. However, the usual benefits that come with the abstraction decision apply here: ease of programming for the common case.

IV. Existing Implementations

This section describes some of the more popular existing implementations of web continuation functionality. The purpose of this section is to present the range of potential designs. This will give context to the new C++ implementation described later in this paper.

```
public void processElement()
{
 int total = 0;
 while (total < 50)
 {
   print(getHtmlTemplate(''form''));
   pause();
   total +=
     getParameterInt(''answer'', 0);
 }

 print(''got a total of ''+total);
}
```

Fig. 5. Example RIFE element to add numbers together until their sum exceeds 50 [14].

A. ViaWeb

Viaweb [6] is likely the earliest example of a continuation based web server. Originating in 1995, Viaweb was an online store system written in Lisp. Its continuations came from the software being written in continuation-passing-style. This gave a clean representation that links and form submissions were tied to the running of some next piece of code.

When a web page was served, the continuation associated with the web page was stored in a global hash table with a unique ID which was embedded within a link on the web page. When the user clicked on this link, the server would find the corresponding continuation and continue the chain of execution.

The creators claim that the main advantage of this style of programming was that they could code interaction with the user as if it was a subroutine call. For the user, it also provided more sophisticated interaction than contemporary web applications.

Viaweb survived, was purchased by Yahoo!, and today is known as the Yahoo! Store. This is testimony to the utility of continuations and beneficial experience for both the developer and user of the resulting system.

B. RIFE

RIFE [14] is an implementation of continuations in pure Java, packaged as a general purpose library. It exposes the continuation via a `pause()` call (see Figure 5).

Multiply restartable continuations are implemented in RIFE by cloning the context of the running element. This puts two notable restrictions on the developer.

First, all entities within the context to be cloned must themselves be clone-able. A workaround to this restriction is provided by RIFE: the developer has the option of specifying that a processing element should simply be suspended and restarted rather than being cloned upon restart. This has the advantage of requiring less resources, but the downside is that the continuation is no longer a solution to back-button issues as it is no longer multiply restartable.

Second, RIFE's particular implementation limits the placement of `pause()` calls to within a `processElement()` method - you may not call `pause()` even in methods called

```
StoreTask >> go
  | cart shipping billing creditCard |
  cart := StoreCart new.
  self isolate: [
    self fillCart: cart
    self confirmContentsOfCart: cart ]
    whileFalse ].
  self isolate: [
    shipping := self getShippingAddress.
    billing :=
      (self useAsBillingAddress: shipping)
      ifFalse: [ self getBillingAddress ]
      ifTrue: [ shipping ].
    creditCard := self getPaymentInfo.
    self ship: cart to: shipping billTo:
        billing payWith: creditCard ].
  self displayConfirmation.
```

Fig. 6. Smalltalk backtracking isolation mechanism [5].

by `processElement()`. The second restriction seems very limiting to the development of modular work-flows.

Storage of continuations in RIFE is via Java's `WeakHashMap`. To avoid resource exhaustion, continuations are destroyed after a timeout has expired (defaulting to twenty minutes).

C. Seaside

Seaside [3], [5] is a Smalltalk framework for developing web applications and has been called "the most popular and influential continuation web server" [15]. Continuations are native to Smalltalk and are exposed to the Seaside programmer via a call-and-answer model and via callbacks associated with form submission.

Seaside also provides a modular abstraction to developers, allowing separate continuation aware components to be combined together in a single page. To address back-button support, Seaside allows components to be registered for backtracking. Only in this case does Seaside persist the state information for multiple restarts of a continuation. The issue of disabling the back-button in certain situations is handled by an isolation structure. Parts of a flow can be bracketed as isolated sections, allowing backtracking within the isolation, but not allowing backtracking into the isolation after its completion (see Figure 6).

Although continuations in Seaside are mediated by an abstraction, complex behaviour can be described by developers due to the modular nature of the Seaside framework and its ability to handle several specific back-button behaviours. All of Seaside's continuations are currently maintained in memory with consideration being given to serializing the continuations to a database in the future.

V. Criticism

Although many new web continuations implementations and libraries are becoming available, there are objections to the use of continuations for web programming.

One of these criticisms comes from Gilad Bracha in his explanation as to why continuations will not be supported natively in Java. He [1] claims continuations are an intermediate technology, like the cassette before the CD. He agrees that the most compelling case for continuations is their use in web applications, but points to AJAX as a sign of the real solution. Regardless of the troubles of implementing native Java continuations with consideration of the Java security model, their passive usefulness is the source of his reservations.

Ian Griffiths [8] goes further to claim that continuations are not even beneficial for use in web applications. His main criticism is that continuations are an inappropriate abstraction that hide important details, specifically the issue of thread affinity. Looking at a content generator written with continuations, it appears as a single function. Generally, one can expect a single function to execute in the same thread from start to finish. However, with the capturing and restarting of continuations, the function will actually be run in different threads throughout its execution. This is the case with fragmented non-continuation based scripts, but it is clear by their fragmentation that they are actually separate execution entities.

Evaluation of this criticism involves answering the question, "What is more important to represent directly: the interaction model, or the execution model?" A continuation based solution cleanly represents the former while obfuscating the latter; the opposite is the case for traditional solutions. If one follows the evolution of programming language abstractions it is certainly the execution model that has lost historically in the struggle for clean representation. Object oriented programming allows provides abstraction using class hierarchies that match our internal representation of the concepts but hides execution details like construction, destruction, dispatch, and implementation of the exposed interface. Abstract oriented programming [9] obfuscates the execution model even further, allowing injection of code at locations across the program, far removed from their actual location of implementation. Yes, one could argue that the inclusion of continuations in web programming is a large jump in abstraction, perhaps as large as the jump to object oriented programming or aspect oriented programming, but it is a jump in the same direction [2].

VI. A C++ Implementation

The remainder of this paper describes an implementation of a continuation web server in C++. It was originally motivated by Graunke et al. [7]. They suggest that any suitably extended high-level language can offer the benefits of continuations to web programming. This C++ implementation was a study to determine what degree of extension was necessary and the degree to which C++ can meet the requirements of this domain. In order to investigate the full set of design decisions, I implemented both the server and two sample programs. The automatic program transformation described by Matthews et al. [11] might prove this work unnecessary, but automating transformation of large templated libraries like the STL or Boost might not be as straight-forward as the examples in Scheme and C that they described. From this study, it becomes clearer how some of the early design decisions regarding the specific implementation of the continuation mechanism constrains later decisions and development possibilities for the web programmer.

A. Content Generators

First, I present an example of the style of programming that this C++ web server allows when developing dynamic content generators. Figure 7 shows a simple content generator that computes the sum of two numbers, fetched sequentially from the user.

This is very close to the style of programming that one could use to target a console as a user interface. There is just slightly more set-up required for the prompt, as it is an HTML page.

B. Implementation Details

The core of the web server is a standard wait-serve loop. Incoming requests are parsed and the appropriate content generator is launched.

For example, in the case of a request for a static page a special content generator for static content is launched. This content generator simply returns either the requested page or a 404 error and exits.

All dynamic content generators and the single static generator derive from an abstract base class `ContentGenerator`. The only pure virtual function that they must implement is the `contentMain()` function. Each content generator must register itself with the `ContentGeneratorFactory` which mediates construction of the content generators. Figure 8 shows a stripped version of the function that handles incoming requests for dynamic content.

Each content generator is actually a thread manager. The `serve()` member function of the content generator starts the content generator's thread, which runs the `contentMain()` function implemented in the derived class. The main server maintains a collection of active content generators. Active content generators are those that have not reported completion and have not timed out.

The `pause()` function is the key to the continuation behaviour. The `pause()` function saves the context and control state of the content generator with an continuation identifier for selective restart (how this is done is described later) and then waits on a condition variable. It takes as a parameter a continuation ID that is randomly created by the `sendPage()` function.

This continuation identifier is embedded in the served form as a hidden field. When the user submits the form, the content generator ID and continuation ID are extracted by the server. The requested content generator is found in the main server's collection and the thread is restarted with the given continuation ID. The context is restored and execution continues at the next instruction after the `pause()` function associated with the restarted continuation.

The store and the control state of the content generator are captured by a combination of saving the content generator's thread's stack and by using the C routine `setjmp()`. This was implemented by extending a small, C, co-routine library by Dan Piponi [12].

When the `mainContent()` function of a content generator returns, a flag is set to indicate its availability for clean-up. A modifiable timeout will also trigger this flag. This garbage collection is performed by a thread running in the main server that iterates over the collection of managed content generators, deleting those that have either completed or timed out.

C. Implications of Design Decisions

It should be clear from previous discussion that subtle design decisions have important implications for the behaviour of a continuation web server. This section outlines where the C++ implementation lies in the space of continuation servers.

This implementation devotes to each interaction session a single thread which is persistent across individual interactions and thus sidesteps Ian Griffith's main criticism of the continuation model [8]. However, there is now the problem of scalability, since the threads are present until their completion or the timeout.

Content generators are collected after their first completion or a time-out. This means that if the user clones the browser window to explore several branches, the first one to complete will invalidate every other continuation from that content generator. The use of a time-out forces the developer of each content generator to make a decision about the appropriate time-out.

Since only slightly more information than the stack beginning at `contentMain()` is saved, there is not much memory overhead associated with holding the context information on the server (however, there is still the overhead of persisting a suspended thread). For example, in the addition example above, the size of the saved stack is only 89 bytes (on Mac OS X).

No information from the heap is saved/restored between restarts of a continuation; dynamically allocated data can not be rewound. It can however, be used as a method of inter-invocation communication.

Embedding the continuation IDs in hidden fields results in ugly URLs which is common to many continuation server implementations. Some users have said they have had difficulty when e-mailing these long url strings [11].

These results are very similar to RIFE. The state is managed by the server, a modifiable timeout value is used, and there are some developer restrictions. In RIFE the restrictions are in the positioning of the `pause()` method call. In this implementation, the restriction is in the inability to rewind heap data.

D. Evaluation

As an informal evaluation, I implemented a slightly more complicated web application on this web server than the addition application: a quiz. The application was to load a set of multiple choice questions from an XML file and display them sequentially to the user. After the user answered all of the

```
#include <boost/lexical_cast.hpp>
#include <string>
#include ``HtmlString.hpp''
#include ``HttpHelper.hpp''
#include ``Addition.hpp''

using namespace std;

Addition::Addition()
{
}

Addition::~Addition()
{
}

void Addition::contentMain()
{
  int sum = getNumber(1) + getNumber(2);

  HtmlString page(``Addition Result'');
  page.beginBody();
  page.addLine(``Sum: `` + boost::lexical_cast<std::string>(sum));
  page.endBody();
  page.endHtml();

  sendData(page);
}

int Addition::getNumber(int which)
{
  HtmlString page(``Addition'');
  page.beginBody();
  page.beginForm(``addition'', ``Addition'');
  page.embedContinuationInfo(getId());
  page.addTextInput(``number'', ``Number ``
    + boost::lexical_cast<std::string>(which) + ``: ``);
  page.addSubmitButton();
  page.endForm();
  page.endBody();
  page.endHtml();

  pause(sendData(page));

  return HttpHelper::getValueFromPath<int>(``number'', m_requestHeader->path(), -1);
}
```

Fig. 7. A content generator for an addition operation.

```
ContentGenerator *cg =
  GeneratorFactory::Instance()->get(getModuleNameFromPath(header->path()));

managedGenerators.push_back(cg);
cg->init(connection, header);
cg->serve();
```

Fig. 8. The dispatch of a dynamic content generator.

questions, their percentage correct was to be displayed. I have developed previously a similar application in Microsoft's ASP, so I had an design in mind when I started programming. The following is a breakdown of the time that I spent developing this application:

- 5 min: Adding a radio group helper function to the HtmlString object
- 15 min: Choosing, downloading and compiling TinyXML, an simple XML library
- 5 min: Writing a sample set of questions in an XML file
- 25 min: Writing the Quiz content generator

At the end of this 50 minutes I had a working quiz web application (see Figure 9). The development time was similar to the time I would have spent if I was targeting a local console rather than a browser. This was much abbreviated from the approximately five hours that I spent writing the ASP version of the same application, a large portion of which was split between learning the specifics of Microsoft's XML parser and setting up the session management mechanism.

This evaluation exposed two advantages of this implementation. The first was that simply having C++ as an option for web programming made available a very extensive collection of libraries, such as TinyXML, that many people would be familiar with already. A second advantage common to any continuation web server is the reduction of development time spent on session management.

There is an outstanding bug in this program however: the back-button does not work. I am still debugging this problem and have narrowed it down to a an invalid memory access. This is an instance of the continuation possibly being an inappropriate abstraction or perhaps C++ being ill-suited to this simulated style of continuation. The saving of the stack and subsequent replacement of the stack via a memcpy operation is quite a significant occurrence to abstract away within a `pause()` call. As of now, all I can say is that there are undocumented restrictions on the developer.

This brings up an interesting question: is it ease of development or the increased functionality of the back-button that is the most important benefit of the continuation? Even though the back-button does not work in my application, I believe that the time saved developing would alone warrant the use of a continuation web server. This gives some validity to RIFE's non-cloneable option for continuations.

VII. Future Work

This work does give optimism that C++ could also be a suitable language for implementation of a continuation web server and associated content generators. This would allow application of the vast amount of C++ code to web programming. However there is much room for improvement on this specific implementation. It would be a great advantage if the state information was not managed at the server. As well, it is a disadvantage that the threads remain suspended between interactions. Future work will focus on remedying these two issues.

There has been no comparison or evaluation of the runtime performance of this implementation. This is another area of interest for future work. A comparison with an automatically transformed C++ program as described in Matthews et al. [11] is one of particular interest.

Lastly, capturing the heap referenced values and restoring them with the continuation would eliminate an anomalous restriction from the developer and truly allow programming for the web as one would program for a local console window.

VIII. Conclusion

Continuations are a useful abstraction for programming the web. Although AJAX may be a sign of an even more rich environment, there will be for a long time a need for the intermediate level of interaction that continuations address.

Although continuations have only recently been applied more broadly to web programming, they have a well developed history and analysis of the associated issues has come into maturity. Where do you store your continuations? How do you decide on a timeout? Do you use a language with native continuations, or simulate continuations in another language? This paper has not provided definite answers to these questions, but has discussed the tradeoffs associated with each of them.

As a proof-of-concept, the implementation of the C++ continuation web server is promising. Further work is yet to be done to refine the implementation and explore alternate design decisions.

```
void Quiz::contentMain()
{
  vector<string> questions;
  vector<int> answerKey;
  vector<vector<string> > options;
  load(questions, options, answerKey);
  int numCorrect = 0;

  vector<string>::const_iterator thisQuestion = questions.begin();
  vector<vector<string> >::const_iterator thisOption = options.begin();
  vector<int>::const_iterator thisAnswer = answerKey.begin();
  for ( ; thisQuestion != questions.end() ; thisQuestion++, thisOption++, thisAnswer++)
    {
      HtmlString page(``Quiz'');
      page.beginBody();
      page.beginForm(``quiz'', ``Quiz'');
      page.addLine(``<p>'' + *thisQuestion + ``</p>'');
      page.addRadioGroup(``answer'', *thisOption);
      page.embedContinuationInfo(getId());
      page.addSubmitButton();
      page.endForm();
      page.endBody();
      page.endHtml();

      pause(sendData(page));

      int answer = HttpHelper::getValueFromPath<int>(``answer'', m_requestHeader->path(), -1);
      if (answer == *thisAnswer)
        {
          numCorrect++;
        }
    }

  HtmlString page(``Results'');
  page.beginBody();
  page.addLine(``<h1>Results</h1>'');
  page.addLine(``<p>'' + boost::lexical_cast<string>(
             100 * (float)numCorrect / (float)questions.size()) + ``%</p>''
             );
  page.endBody();
  page.endHtml();
  sendData(page);
}
```

Fig. 9. The *contentMain()* function of the Quiz content generator.

REFERENCES

[1] G. Bracha, *Will Continuations continue?* [Online]. Available: http://blogs.sun.com/gbracha/entry/will_continuations_continue

[2] A. Bryant, "Mit artificial intelligence lab lightweight languages workshop mailing list." [Online]. Available: http://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg02476.html

[3] A. Bryant and The Seaside Community. [Online]. Available: http://www.seaside.st/

[4] M. Burns, G. Pettyjohn, and J. McCarthy, *PLT Scheme Web Server Manual*. [Online]. Available: http://download.plt-scheme.org/doc/352/html/web-server/

[5] S. Ducasse, A. Lienhard, and L. Renggli, "Seaside - a multiple control flow web application framework," in *European Smalltalk User Group Conference (EUSG)*, 2004, pp. 231–254.

[6] P. Graham, "Lisp for web-based applications." [Online]. Available: http://www.paulgraham.com/lwba.html

[7] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen, "Programming the Web with high-level programming languages," *Lecture Notes in Computer Science*, vol. 2028, pp. 122–??, 2001.

[8] I. Griffiths, "Continuations for user journeys in web applications considered harmful." [Online]. Available: http://www.interact-sw.co.uk/iangblog/2006/05/21/webcontinuations

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 1997.

[10] S. Krishnamurthi, "The continue server (or, how i administered padl 2002 and 2003)," in *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*. London, UK: Springer-Verlag, 2003, pp. 2–16.

[11] J. Matthews, R. Findler, P. Graunke, S. Krishnamurthi, and M. Felleisen, "Automatically restructuring programs for the web," *Automated Software Engineering Journal*, 2004.

[12] D. Piponi, "Continuations in c." [Online]. Available: http://homepage.mac.com/sigfpe/Computing/continuations.html

[13] C. Queinnec, "Continuations and web servers," *Higher-Order and Symbolic Computation*, vol. 17, pp. 277–295, 2004.

[14] RIFE, http://rifers.org.

[15] B. Tate, *Crossing borders: Continuations, Web development, and Java programming*. [Online]. Available: http://www-128.ibm.com/developerworks/web/library/j-cb03216/index.html