An Infernal Device

Browser-Stored Do-It-Yourself Continuations for Web Programming

Tommy M. McGuire
The University of Texas at Austin
mcguire@cs.utexas.edu

ABSTRACT

One of the greatest difficulties of writing web-based applications is the stateless nature of the Hyper-Text Transport Protocol. In particular, the stateless protocol makes the control structure of the application much less clear. One excellent approach to handling the control structure problem is the use of *continuations*, data structures which explicitly represent the control structure of a program. This paper presents a framework for designing web applications, based on continuations, which significantly clears up the control structure of web applications without requiring specialized languages or tools. It also discusses the *kinds* of storage available in the web server/web browser system and argues that different parts of the state of the application, including the control state, belong in particular kinds of storage.

1. INTRODUCTION

Consider the program fragment in Figure 1, similar to examples from Graunke, Findler, Krishnamurthi, and Felleisen[5], and Queinnec[12]. This program fragment prompts for two numbers, adds them, and displays the result. This fragment would normally be considered a basic programming exercise, but when seen in the context of a web application becomes considerably more complex.

n = prompt-read ("Enter first number");
m = prompt-read ("Enter second number");
display $(n + m)$

Figure 1: A simple program

The difficulty comes from the stateless nature of the HTTP communications protocol—when receiving a request, the program "starts fresh," with no idea of where it is in the computation. The difficulty is greatest in the most common, simplest web application environment—the Common Gateway Interface (or CGI). In this environment, application programs are executed in response to HTTP queries based solely on the URL and must generate a new response based on input from the query. Once a response is generated, the application program must then exit.

WWW2004, May 17–22, 2004, New York, NY USA.
ACM xxx.xxx.

The typical response to the difficulty is typically ad-hoc, using multiple programs responding to different URLs or a single program that attempts to determine its location in the computation based on its inputs. Ad-hoc approaches usually wind up both being unforgivably complex and giving strange results when presented with out-of-order HTTP queries, multiple simultaneous submissions from several browser windows, and the use of the browser's infamous Back button. (Indeed, as far as the user is concerned, the Back button returns the computation to a previous state, although the program may think otherwise).

The best response to the difficulty allows the programmer to write the application in a direct style, as in Figure 1, and provides infrastructure for automatically handling that style in the web environment. This response has been explored by Queinnec[13, 12], Graunke, et al.[7, 5], and Thiemann[14, 15]. Most of this research uses *continuations*, which are normally described as data structures representing "the rest of the computation," or at least the state of the control stack of the computation. (*Essentials of Programming Languages*[3] contains more information on continuations.)

To illustrate the use of continuations in web applications, Figure 2 shows pseudocode for the function prompt-read. In this figure, prompt-read is called with a prompt string and immediately captures the current continuation as k, describing what the application wishes to do with the resulting value of prompt-read. The function preserves the continuation, associates it with a HTML fragment such as a URL or a form field, and sends a response HTML page containing the prompt text and the HTML fragment. Then the program terminates.

```
function prompt-read (prompt) =
  begin
    k = capture-continuation();
    frag = continuation-to-html(k);
    print-html(prompt, frag);
    exit
  end
```

Figure 2: A pseudocode prompt-read

This prompt-read function needs to be matched with a driving part of the web application, a dispatch function as

in Figure 3. When the server receives a request, the application attempts to determine if it contains a continuation, such as the one from the HTML fragment from the page produced above. If so, the continuation should be invoked with the contents of the request as its arguments. Otherwise, a default continuation is invoked with the request. In either case, invoking a continuation with a value is equivalent to returning the value from the function which originally called capture-continuation, here prompt-read.

```
function dispatch (request) =
  begin
    if has-continuation(request)
      then k = recover-continuation(request)
      else k = get-default-continuation();
    invoke-continuation(k, request)
  end
```

Figure 3: Pseudocode for the dispatch function

The continuation of the first call to prompt-read in Figure 1 is something like Figure 4—the value entered from the request is accessed as n, the program prompts for and reads a second value, m, and displays $n + m$.

```
function first-continuation (response) =
  begin
    n = parse-http (response);
    m = prompt-read ("Enter second number");
    display(n + m);
  end
```

Figure 4: A pseudocode continuation of the adder, with some hand waving

(Unfortunately, continuations in general are not anywhere near as simple as Figure 4. They not only must encapsulate what to do next, but must caputure the local storage of the application as well. For example, the continuation captured by the second call to prompt-read from Figure 4 must contain the value of n as well as the information that the result of $n + m$ should be passed to display.)

Many of the systems described in previous research[13, 12, 7, 5, 14, 15] have one or more of the following weaknesses:

1. Most languages do not have first-class continuations, and many language implementations that have them do not easily allow them to be stored and recovered between executions of the program. Thus, the work based on first-class continuations (particularly [13, 12, 7]) is limited to languages supporting them and specialized web servers written in that language. Also, if the continuations cannot be stored, the application may have problems scaling since it will face more difficulties in being spread across a cluster of server machines.

2. One solution to the lack of first-class, storable continuations is to translate the program to continuation-passing style (or CPS) (as in [5, 8]). This translation creates data structures representing continuations and adds an argument to each function allowing the correct continuation to be passed where it is needed. (For more information on CPS, see *EoPL*[3], as well.) Translating a program to CPS by hand is difficult and error-prone, and therefore an approach based on it requires a pass through a language translator, something else that may not be available for many languages. Also, using the continuation-passing style in any language that does not have tail-call optimization requires complex techniques to prevent the stack from growing with every continuation invocation.

3. A continuation, resulting from either a language's support or from a translation to continuation-passing style, may be large. The direct style of programming does not provide the programmer either direct control of or information about the size of the continuations.

4. Most of these systems store the continuations locally on the server, with some kind of token identifying the continuation stored in the page sent to the browser. In general, the server is not in a position to know when a continuation may be deleted—the continuation may be invoked days or weeks after being created.

This paper presents another attempt at resolving those pesky weaknesses by noting that:

- The only time a continuation needs to be captured is when a response is being sent to the browser to display a page,
- These continuations are not necessarily complex if the application is structured properly and its state is managed well, and
- The proper use of encryption can be used to allow the browser to store much of the state of the application.

Section 2 describes this attempt in detail, and Section 3 presents a simple application of it. Section 4 discusses several issues raised by the attempt. Section 5 attempts to describe the kinds of storage the web application has at its disposal and the kind of application state best suited to each. Finally, Section 6 relates this research to its context and Section 7 concludes the paper.

2. AN INFERNAL DEVICE

Figure 5 presents the page flow for the simple adder application (such as Figure 1)—the sequence of pages seen by the user. The only places where the continuations can be invoked in this page flow are at the sources of the arrows between pages, as the processing of the HTTP query. The only places where the continuations described above need to be captured are the destinations of the arrows, immediately before a page is generated and the program terminates. These

continuations in a web application can be particularly simple since they are only used to store and invoke the immediate next step of the computation.

The structure of the page flow of web applications means that the application can be viewed as a state machine, as in Figure 6. In this state machine, application computation takes place during transitions and the states represent user-visible web pages.

The infernal device inverts the normal view of such a state machine: transitions are represented by data structures.[1] These data structures must have one operation, a method to perform the transition's computation. The data structures need to be stored after the termination of the previous transition, in order to be invoked when needed. In a nutshell, these transitions are (a poor approximation of) continuations. They are, however, capable of being serialized to a secure text form that can be given to the browser as part of a URL or as a field in a HTML form. With such stored transitions, the user can return to a previous page and continue the computation with different inputs or take multiple paths simultaneously in different browser windows.

The object-oriented framework upon which this work is based is implemented in PHP and has a small number of classes: Transition, TransitionKey, and Machine. Although PHP is used here, any language will provide greater or lesser support for the technique; object-oriented languages that support marshaling objects to strings allow this approach easily, without modification, but languages which do not support marshaling objects require only a little more work.

The Transition is the base class of continuations representing the steps of the computation. Each descendent of this class has at least one method, execute, which takes as arguments a reference to the application's state machine and a HTTP request. This method can perform any necessary computation, but should do at least two things:

- Create any outgoing transitions from the next state.[2]
- Print out the HTML page forming the next state.

When an outgoing Transition is created, it is passed to the get_transitionkey method of the Machine class, which returns a TransitionKey containing the *bonded* Transition. When a Transition is bonded,

1. The Transition is serialized to a text form suitable for external storage.
2. The serialized Transition is compressed.
3. A message integrity check, or MIC, is computed for the Transition.
4. The compressed Transition and the MIC are concatenated and encrypted.
5. The encrypted string is encoded in Base-64, resulting in a simple text string.

The Transition can then be safely stored in a URL (if it is not too large) or as part of a HTML form on a web page. The TransitionKey is used to format either a URL query element or a hidden form field as needed.

The Machine represents the state machine itself. Each application should create a subclass of Machine, with at least a method called initial_transition, returning an instance of the application's initial Transition. This method is called whenever no other Transition can be identified from the HTTP request and provides the default transition. No Transition will be recoverable either if the request is the first request to an application or if the decryption of a Transition fails.

A single instance of this subclass of Machine should be created by the application when it is initially invoked to respond to a web request. The Machine's transition method should then be called with the request to *release*[3] the bonded Transition and take the next step in the application.

3. AN APPLICATION: ADDING TWO NUMBERS

The framework presented here is simple, and probably simplistic, but it does provide a tolerable way of structuring a web application in addition to dealing with some of the persistent behavioral irregularities of many web programs. To demonstrate the structure, this section presents, as an example, part of the adder application described in Section 1. This example is written in PHP, using the Machine module and no other extensions.[4]

In order to produce a good user interface, the page flow of an application (such as Figure 5) should be created first and from that a state machine for the application can be created (as in Figure 6, although it should include error handling as discussed in Section 4 and shown in Figure 9). From the state machine, a programmer can create skeletal Transitions, leaving only the processing in the Transitions' execution methods.

The initial transition from Figure 6 is presented in Figure 7. An instance of the InitialTrans class will be created by the initial_transition method of the application's Machine instance and immediately executed the first time the application is visited. Executing an InitialTrans creates an instance of the GotFirst Transition subclass (see Figure 6 for the relationship between the two transitions and the state between them) and then displays the HTML page with the form asking for the first number. Since this is the initial transition, the request has no information for the application, and is not examined.

Further along in the application, Figure 8 shows the last transition of Figure 9, including the additional validation

[1]It also turns out to be quite useful to represent the states as functions, as will be seen shortly.

[2]A Transition leading to a terminal state does not actually create any outgoing transitions, but the only way off of such a page and out of such a terminal state is either the browser's navigation controls or normal HTML hyperlinks.

[3]Picture the inverse of the bonding steps above.

[4]Except for a basic Lame Template System, or LTS, to produce the HTML output. The LTS is included in the distribution with the Machine module. Any other PHP system can also be used.

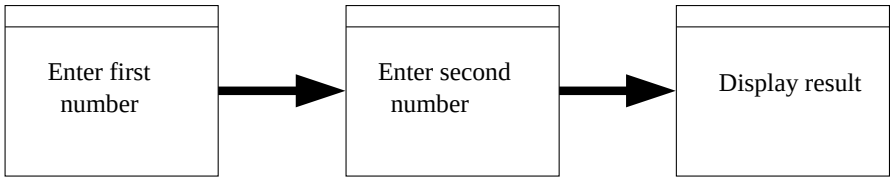

Figure 5: Page flow for the adder

```
class InitialTrans extends Transition
{
  function execute ($machine, $request) {
    $trans = new GotFirst();
    $k = $machine->get_transitionkey($trans);
    show_first($machine, $k,
    '<p>Please enter a number.</p>', '0');
  }
}
```

Figure 7: The initial transition

discussed below. When an instance of the GotSecond Transition is created as part of the GotFirst Transition, the first value entered by the user is passed to GotSecond's constructor and stored in a field. When the GotSecond Transition is released, the second value is picked from the HTTP request. The Transition displays the two numbers and the addition, and enters a final state with no outgoing transitions. The final state can be exited by following any hyperlink on the page or by using the browser's navigation controls.

One further point to note in Figure 8 is the storage of the first number entered, n. Unlike the direct style of programming discussed in Section 1, the programmer has direct control over the size of the stored Transitions by limiting or expanding the fields of the Transition. On the other hand, the programmer has to take control over the fields of the

```
class GotSecond extends Transition
{
  var $n;
  function GotSecond ($n) { $this->n = $n; }
  function execute ($machine, $request) {
    $m = trim($request['number']);
    if (is_valid($m)) {
      show_addition($machine,
      $this->n, $m,
      $this->n + $m);
    } else {
      $k = $machine->get_transitionkey($this);
      show_second($machine, $k,
      '<p>Please try again.</p>',
      $this->n, $m);
    }
  }
}
```

Figure 8: The final transition

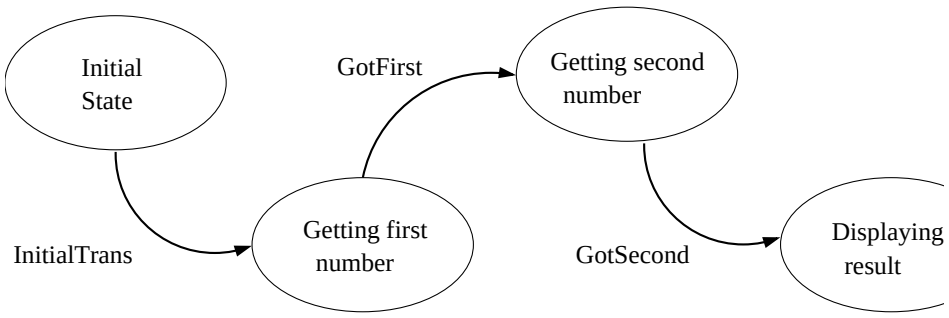

Figure 6: A state machine for the adder

stored Transitions, which adds to the complexity.

While these examples appear verbose, perhaps mostly due to the syntax of PHP, it is important to remember that these transitions are almost all there is to creating a web application—the only remaining components of this application are the Machine subclass, to set the initial transition and deal with the encryption and integrity keys, and the templates and other presentation elements.

4. OTHER ISSUES

This section discusses a number of more subtle issues raised by the framework.

4.1 Transitions with multiple destinations

Hypothetically, suppose the user sends a request containing "AString" in response to the second input state in the application of Figure 6. The application should detect that the input is not a number and do something appropriate. (Thiemann[14, 15] uses the Haskell type system to protect the application from this sort of error, but the framework described here is far less complex and relies on dynamic typing.)

One possibility is to return to the previous state, issuing a warning message, and re-prompting for the second number, as in Figure 9. Figure 8 is an example of a Transition that does just that, reusing the instance of GotSecond as the stored, outgoing transition. Fortunately, there is no need for a Transition to have only a single destination—since the states are user-visible pages, the destination state depends entirely on what the Transition prints.

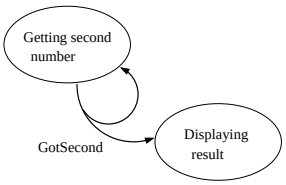

Figure 9: A Transition with multiple destination states

The possibility of Transitions with multiple destination states is the reason that the states of the application should be represented in the program by separate functions, rather than ad-hoc code to generate the HTML pages from within Transactions. In this specific example, both the GotFirst and the GotSecond transactions need to be able to go into the "Enter second number" state, which is greatly eased by separating the code that creates that state (by printing out the "Enter second number" web form) into a function.

4.2 Transactions

Another potentially problematic issue is preventing a Transition from being taken repeatedly when it should only be taken once. Queinnec's work[13, 12] provides one solution by creating continuations in prompt-read that can only be invoked once. This framework requires more work, specifically a bit of coordination between the stored Transitions and the server's persistent state. One form of coordination is to store a transaction identifier in the Transition and to invalidate that transaction identifier once the transaction is committed or canned. The code for the Transaction can check the validity of the transaction identifier and present an error message if necessary.

4.3 Security

While the infernal device is not quite trustingly executing *code* it gets from a web browser, it is executing with trusted *data* from the browser. There are two issues involved in storing the Transitions remotely:

- Integrity—the data that the state machine uses from the remotely stored Transition must be the same data that the machine stored there. To do otherwise invites abuse.
- Confidentiality—the data that the state machine stores with the browser may contain information that the browser's user should not have access to. Failing to guarantee that this information will remain confidential invites severe contortions on the part of the application programmer.

Unfortunately, the web application environment is antagonistic to these needs. For example, a Transition may be stored and invoked long after it was created. As well, replay attacks are what this framework is specifically trying to allow—that is what the browser's Back button does.

In order to ensure confidentiality, during the bonding process the framework encrypts the serialized Transition; because the environment calls for both strong and fast encryption, the default algorithm chosen is Rijndael, the basis of the Advanced Encryption Standard, with a 128-bit key. The algorithm is used in cipher block chaining mode, to avoid problems with repetitive information in the serialized Transition. The encryption key (as well as the integrity key, described below) is a parameter of the Machine class constructor. For flexibility, the algorithm and encryption mode can be changed by the application's subclass of Machine to any other supported by PHP's mcrypt module.

The bonding process also includes a message integrity check, computed using a protocol described in Section 5.2.2 of [10]. A cryptographic hash is computed of the serialized Transition concatenated with a secret integrity key. By default, the MD5 algorithm is used to compute the hash, although this also can be changed by the application's Machine subclass.

4.4 Authentication and authorization

Two issues related to security are that of authentication and authorization. Some web applications need to identify the user on the other end of the browser; others merely need to make sure that whoever is there *should* be doing whatever the application does. For the purposes of this framework, both issues are treated identically, by not being treated at all.

The Transitions created by this device can be bookmarked if presented in a URL, stored if presented in a form field, or simply copied in either case. None of these operations are desirable for authentication or authorization information, which should therefore not be stored in Transitions. (As discussed in Section 5, browser-local page-independent storage is probably the best option for these items.)

5. A DIGRESSION ON WEB STORAGE

The web server/web browser system supports four different *kinds* of storage:[5]

1. Server-side storage includes information kept in the web server associated with the application. Server-side storage can be either transient or persistent:
   (a) Transient storage is generated and used during the processing of one web request. It is not preserved between requests and must be recreated if needed in a subsequent request. Transient storage is generally required for file storage descriptors, database connections, and similar resource handles used in processing requests.[6]
   (b) Persistent storage includes everything on the web server that is not local to the computation responding to a single request. This includes the state of the server itself (to the extent that such information is useful to the web application) as well as data contained in the server's filesystem or a related database.
2. Browser-side storage includes information sent to the browser from the application, and is therefore persistent between requests. Browser-side storage can be either page-local or page-independent:
   (a) Page-local storage includes information encoded in URLs and in form fields embedded in HTML pages. This information is stored by the browser and returned to the server in a HTTP request resulting from following the URL link or submitting the form.

[5]There are possibly five kinds, adding client-side transient storage when in the presence of browser-executed scripts. This analysis is getting complex enough, though.

[6]Many systems are capable of caching resource handles such as database connections between requests, but these caching systems are logically identical to using transient resource handles—the application must acquire the handle from the cache in much the same way as it would acquire a handle initially. As well, many systems such as vanilla CGI programs are not capable of such caching.

(b) Page-independent storage includes information encoded in cookies sent to the browser by the web application. This information is also returned to the server as part of HTTP requests, but the information is not related to the contents of the URL or form submitted—the last information sent will be returned to the server in all cases.

Many of the problems seen by users with web applications are based in confusion on the part of the application programmer about the different kinds of storage. (Graunke, Findler, Krishnamurthi, and Felleisen[6] provide an excellent framework for the analysis of these problems, although they seem not to mention browser-side page-independent storage. On the other hand, such storage can be simulated in their model by combining page-local storage and server-side persistent storage.) Understanding the proper place of application state in the different kinds of storage is important while designing web applications.

In the first place, server-side persistent storage should be avoided for anything that can avoid it. It is better to regenerate information or hand it to the browser if at all possible. Failing to do this may slow the web server and can lead to either unbounded resource consumption or arbitrary limitations on application behavior, since the server generally has no idea when it can safely remove such information or when the information must be kept because some client is about to use it.

On the other hand, server-side persistent storage is necessary for information that must be shared between clients, such as airline reservations, web log entries, game high scores, and so on.

Page-local storage is the ideal location for information such as the stored Transitions provided by this framework, which makes browser-side page-local storage much easier to access for this use. From the user's viewpoint, whether it is the result of a server's most recent response or the action of the browser's navigation tools, the page displayed by the browser represents the current state of the application, and the next action taken by the application in response to the user's behavior should be the result of information on the currently displayed page. Such state should be kept small, but extreme measures are not necessary since the contents of a page can be relatively large and the browser is in a position to delete anything no longer needed. The presence of transactions complicates the situation, but not greatly since the user should certainly be aware of transactional limits in an application. Likewise, shared information held in server-side persistent storage could also be a problem but the user should be aware of interactions with other users (or other browser windows, for the multitasking user).

Page-independent storage is useful only for session-like information such as:

- The state of a game (such as guess-the-number from Thiemann[15]) which would be incorrect if stored page-locally and is not shared between users), and
- Authentication and authorization information.

Given the stateless nature of the HTTP protocol, page-independent storage is the best option, since cookies that are stored only for a single execution of the browser or for only a limited time contain the damage possible given this storage, and are tolerably understandable from the user's viewpoint.

6. RELATED WORK

Paul Graham[4] described the benefits of manually using a continuation-passing style while writing web applications.

John Hughes[9] described the use of an extension of monads in Haskell, in part to remotely store the state of a web application. This system is one alternative to continuations.

Christian Queinnec[13, 12] described the fundamental basis of using continuations to support programming web applications in a direct style. The work includes approaches to dealing with transactions, computations that fail to return a page speedily, and computations that return multiple pages. Queinnec mentions the possibility of storing the continuations in the browser, although the work uses a web server written in Scheme and dealing with locally-stored continuations. The second paper[12] describes continuation sizes of 100KB, although mentioning the possibility of reducing them to "a few Kbytes or a few bytes if the web application is considered constant." The continuations seen with the framework described in this work bear out the latter possibility, being in the range of tens to hundreds of bytes.

Graunke, Krishnamurthi, Van Der Hoeven, and Felleisen[7], like Queinnec, present a web server written in Scheme with locally-stored continuations. The server keeps the continuations for a predetermined lifetime as well as providing a function to terminate a computation and remove its stored continuations. Also, the continuations can only be resumed by the original thread that created them.

Graunke, Findler, Krishnamurthi, and Felleisen[5] also describe using the transformation to continuation-passing style in the Scheme programming language (but without using Scheme's advanced control structures), and serializing the continuation to the browser's storage. The paper mentions the security issues of storing the continuations remotely, as well. This paper is easily the most closely related to the current work, ending with the observation that,

> Even in the absence of [tools to facilitate the continuation passing style], programmers can achieve a lesser degree of benefit by proceeding in a systematic manner and documenting the design pattern.

The framework described in this paper is, in effect, a system for doing exactly that.

Graunke and Krishnamurthi[8] extend the use of the continuation-passing style to the Java programming language and argues that the web browser's navigation ability could usefully be added to local GUI applications.

Thiemann[14, 15] presents a much more comprehensive system, including a page generation language, server-side and browser-side storage management, and type safety for all aspects of the web application. This system does store the computation state on the server side, but does not use continuations to do so—instead, it stores a log of the inputs to the application, both HTML forms and local I/O. When invoked, the web application replays the stored inputs and its own computation up to the point where further input is needed from the browser, at which point the log is stored again. WASH/CGI is a fascinating and different system, but the possibility of compute-intensive applications would seem to make it impractical.

Manolescu[11] develops a similar approach in a different context, that of a workflow framework. The workflow framework separates control flow from application processing where the application and its control flow may be extremely long-lived. Like the framework described here, Manolescu argues that techniques from programming language research provide solutions in other, apparently unrelated, fields.

Apache Cocoon 2.1[1], a web publishing framework, uses server-side stored continuations from JavaScript for flow control in applications. Likewise, Seaside[2] uses continuations from Smalltalk. Both appear to store the continuations on the server, but unfortunately, I have not examined either closely.

7. CONCLUSION

This paper presents an infernal device for writing web applications in languages that do not support first-class, storable continuations and without going through a full translation to the continuation-passing style. Although somewhat rough around the edges, the system appears to be scalable and provides the programmer direct control over the location and size of the state of the application.

The paper also digresses into the nature of the web server/web browser system, particularly regarding the kinds of storage available for information between the two. This digression included tips on what kind of state needed to be put where.

However, the primary contribution of this paper is to explore the use of advanced programming language constructs, continuations, in the context of a common application without requiring the use of an advanced programming language. Continuations make web programming both simpler and more robust, and when renamed as transitions, both apply to any programming environment and do not introduce excess complexity.

Further work on this system includes extending it to other languages and examining the performance of the system, particularly given the encryption and integrity checking steps. (It is interesting to note that the alternative to encryption is to store the Transition locally, resulting in a comparison between the speed of the processor doing encryption and the I/O accessing persistent local storage.)

Another extension of the system that should be examined is the use of a program generator to automatically create the boilerplate of the Transitions. It is even possible that this program generator would be language-agnostic, by leaving the code within a transition untouched. Such a generator would be much simpler than a translator using the continuation-passing style.

In any case, the approach described here, as well as those discussed above, provide an excellent way of designing web-based applications, while also preserving the usefulness of the browser's navigation interface.

The software described in this paper is available from: http://www.cs.utexas.edu/users/mcguire/software/id/.

8. REFERENCES

[1] Apache cocoon 2.1, August 2003. http://cocoon.apache.org/2.1/.

[2] Seaside, August 2003. http://www.beta4.com/seaside2/.

[3] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, second edition, 2001.

[4] Paul Graham. Lisp for web-based applications, August 2003. http://www.paulgraham.com/articles.html.

[5] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the web. In *IEEE International Conference on Automated Software Engineering*, November 2001.

[6] P. Graunke, R. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions, 2003.

[7] Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with high-level programming languages. *Lecture Notes in Computer Science*, 2028, 2001.

[8] Paul T. Graunke and Shriram Krishnamurthi. Advanced control flows for flexible graphical user interfaces or, growing guis on trees or, bookmarking guis, 2002.

[9] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37, May 2000.

[10] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security, PRIVATE Communication in a PUBLIC World*. Prentice Hall, second edition, 2002.

[11] Dragos A. Manolescu. Workflow enactment with continuation and future objects. In *OOPSLA '02*, nov 2002.

[12] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming.

[13] Christian Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. *ACM SIGPLAN Notices*, 35(9):23–33, 2000.

[14] Peter Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *Practical Aspects of Declarative Languages*, pages 192–208, 2002.

[15] Peter Thiemann. An embedded domain-specific language for type-safe server-side web-scripting. http://www.informatik.uni-freiburg.de/ thiemann/-haskell/WASH/draft.pdf, August 2003.

History

July 21, 2003. Original draft completed.

August 20, 2003. Cocoon and Seaside references added. Section 1 of the original draft was also confused as to what part of the application state is saved in a continuation.

November 14, 2003. Added workflow reference, reformatted and edited for submission to WWW2004.