Interaction-Safe State for the Web

Jay McCarthy   Shriram Krishnamurthi

Brown University

jay@cs.brown.edu   sk@cs.brown.edu

Abstract

Recent research has demonstrated that continuations provide a clean basis to describe interactive Web programs. This account, however, provides only a limited description of state, which is essential to Web applications. This state is affected by the numerous control operators (known as navigation buttons) in Web browsers, which make Web applications behave in unexpected and even erroneous ways.

We describe these subtleties as discovered in the context of working Web applications. Based on this analysis we present linguistic extensions that accurately capture state in the context of the Web, presenting a novel form of dynamic scope. We support this investigation with a formal semantics and a discussion of applications. The results of this paper have already been successfully applied to working applications.

1. Introduction

The Web has become one of the most effective media for software deployment. Users no longer need to download large run-time systems, and developers are free to use their choice of programming language(s). Web browsers have grown in sophistication, enabling the construction of interfaces that increasingly rival desktop applications. The ability to centralize data improves access and reliability. Finally, individual users no longer need to install or upgrade software, since this can be done centrally and seamlessly on the server, realizing a vision of always up-to-date software.

Set against these benefits, Web application developers must confront several problems. One of the most bothersome is the impact of the stateless Web protocol on the structure of the source program. The protocol forces developers to employ a form of continuation-passing style, where the continuation represents the computation that would otherwise be lost when the server terminates servlet execution at each interaction point. Recent research demonstrates that using continuations in the source reinstates the structure of the program [13, 14, 16, 22].

Another source of difficulty is the Web browser itself. Browsers permit users to perform actions such as cloning windows or clicking the Back button. These are effectively (extra-linguistic) control operators, because they have an effect on the program's control flow. The interaction between these and state in the program can have sufficiently unexpected consequences that it induces errors even in major commercial Web sites [18]. The use of continuations does not eliminate these problems because continuations do not close over the values of mutable state.

One solution to this latter problem would be to disallow mutation entirely. Web applications do, however, contain stateful elements—e.g., the content of a shopping cart—that must persist over the course of a browsing session. To describe these succinctly and modularly (i.e., without transforming the entire program into a particular idiomatic style), it is natural to use mutation. It is therefore essential to have mutable server-side state that accounts for user interactions.

In this paper, we present a notion of state that is appropriate for interactive Web applications. The described *cells* are mutable, and follow a peculiar scoping property: rather than being scoped over the syntactic tree of expressions, they are scoped over a dynamic tree of Web interactions. To motivate the need for this notion of state, we first illustrate the kinds of interactions that stateful code must support (Sec. 2). We then informally explain why traditional state mechanisms (that are, faultily, used in some existing Web applications) fail to demonstrate these requirements (Sec. 3) and then introduce our notion of state (Sec. 4) with a semantics (Sec. 5). We also briefly discuss applications (Sec. 6) and performance (Sec. 7) from deployed applications.

2. Motivation

The PLT Scheme Web server [14], a modern Web server implemented entirely in Scheme, is a test-bed for experimenting with continuation-based Web programming. The server runs numerous working Web applications. One of the most prominent is CONTINUE [15, 17], which manages the paper submission and review phases of academic conferences. CONTINUE has been used by several conferences including Compiler Construction, the Computer Security Foundations Workshop, the ACM Symposium on Principles of Programming Languages, the International Symposium on Software Testing and Analysis, the ACM Symposium on Software Visualization, and others.

CONTINUE employs a sortable list display component. This component is used in multiple places, such as the display of the list of the papers submitted to the conference. The component has the following behaviors: the sort strategy may be reversed (i.e., it may be shown in descending or ascending order); when the user presses the Back button after changing the sort, the user sees the list sorted as it was prior to the change; the user may clone the window and explore the list with different sorts in different browser windows, without the sort used in one window interfering with the sort used in another; and, when the user returns to the list after a detour into some other part of the application, the list remains sorted in the same way as it was prior to the detour, while its content reflects changes to the state (e.g., it includes newly-submitted papers).

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

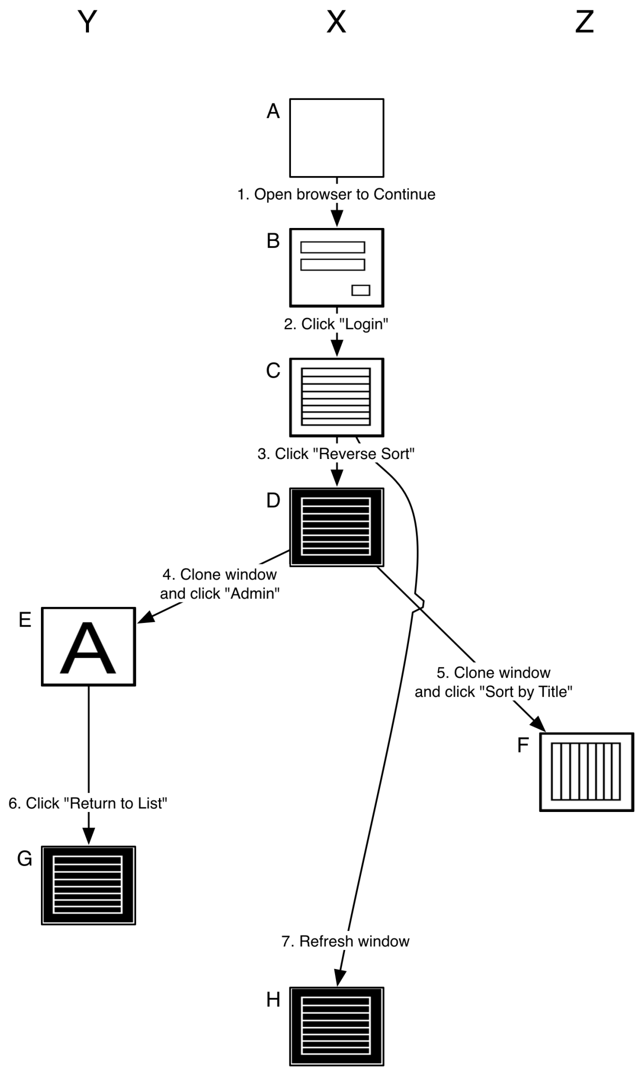
Y
X
Z
A
1. Open browser to Continue
B
2. Click "Login"
C
3. Click "Reverse Sort"
D
4. Clone window and click "Admin"
E
A
5. Clone window and click "Sort by Title"
F
6. Click "Return to List"
G
7. Refresh window
H

Figure 1. A diagram of an interaction with CONTINUE

Let us consider a sequence of interactions with the list of papers, and examine how we would expect this component to behave. Fig. 1 presents the outcome of these interactions. We use these interactions as our primary example throughout the paper, so it is important for the reader to study this scenario. Links in the pages that are produced by user actions are represented as links in the tree structure of the diagram. Because these actions produce multiple browser windows, the diagram has three columns, one for each browser window. Because these actions have a temporal ordering, the diagram has eight rows, one for each time step.

In this diagram, the user's browser windows are labeled X, Y, and Z. The diagram uses icons to represent the content of pages. A window with horizontal stripes represents a list sorted by author, while one with vertical stripes represents a list sorted by title. The color-inverted versions of these windows represent the same sort but in the reverse order. The 'A' icon represents the site administration page.

In Fig. 1, each node represents the page associated with a URL the user visits. A node, i, is the child of another node, j, when the page of j contains the URL of the page of i in a link or the `action` attribute of a form, and the user followed the link or submitted the form. Each edge is labeled with the action the user performed. The numbers on the edges indicate the temporal order of the actions.

When reading this diagram, it is important to recall that some user actions are not seen by the server. For example, action 4 creates a new window and then follows a link. The server is not notified of the cloning, so it only sees a request for the administrative section; presenting the generated content in window Y is the browser's responsibility.

This diagram contains some interesting interactions that highlight the requirements on the list component. First, the user clones the browser windows, which tests the facility to keep a separate and independent sort state for each browser window. This is equivalent to ensuring that the Back and Refresh buttons work correctly [20]. Second, the sort state is not lost when the user goes to a different part of the site (e.g., the Admin section in action 4) and then returns (action 6) to the list.

The placement of node H in Fig. 1 needs particular explanation. The edge leading to this node (7) is labeled with a Refresh action. Many users expect Refresh to "re-display the current page", though they implicitly expect to see updates to the underlying state (e.g., refreshing a list of email messages should display new messages received since the page was last generated). Even some browsers assume this, effectively refreshing the page when a user requests to save or print it. Under this understanding, the action 7 would simply redisplay node D.

In reality, however, the browser (if it has not cached the page) sends an HTTP request for the currently displayed URL. This request is indistinguishable from the first request on the URL, modulo timestamps, causing program execution from the previous interaction point.[1] Therefore, when the user performs action 7, the server does not receive a "redisplay" request; it instead receives a request for the content pointed to by the 'Reverse Sort' link. The server dutifully handles this request in the same way it handled the request corresponding to action 3, in this example displaying a new page that happens to look the same, modulo new papers.

Now that we have established an understanding of the desired interaction semantics of our component, we will describe the problem, introduce the solution context and then describe the solution.

3. Problem Statement and Failed Approaches

We have seen a set of requirements on the list component that have to do with the proper maintenance of state in the presence of user interactions (as shown in Fig. 1).These requirements reflect the intended state of the component, i.e., the current sort state: ordering (ascending vs. descending) and strategy (by-author vs. by-title).

We observe that values that describe the display of each page are defined by the sequence of user actions that lead to it from the root. For example, node G represents a list sorted by author in reverse ordering, because action 2 initializes the sort state to "sort by author", action 3 reverses the sort, and actions 4 and 6 do not change the sort state. To understand that the same holds true of node H, recall the true meaning of Refresh discussed earlier.

These observations indicate that there is a form of state whose modifications should be confined to the subtree rooted at the point of the modification. For example, action 4's effect is contained in the subtree rooted at node E; therefore, action 5 and node F are unaffected by action 4, because neither is in the subtree rooted at E. The challenge is to implement such state in an application.

[1] This execution can produce a drastically different outcome that would not be recognized as the "same" page, or at the very least can cause re-execution of operations that change the state: this is why, on some Web sites, printing or saving a receipt can cause billing to take place a second time.

Context

To discuss the various types of state that are available for our use, we present our work in the context of the PLT Scheme Web server. This server exposes all the underlying state mechanisms of Scheme, and should thus provide a common foundation for discussing their merits.

The PLT Scheme Web server [14] enables a direct style of Web application development that past research has found beneficial. This past research [13, 14, 16, 22] has observed that Web programs written atop the Web's CGI protocol have a form akin to continuation-passing style (CPS). A system can eliminate this burden for programmers by automatically capturing the continuation at interaction points, and resuming this captured continuation on requests.

The PLT Scheme Web server endows servlet authors with a key primitive, **send/suspend**. This primitive captures the current continuation, binds it to a URL, invokes an HTML response generation function with that URL to generate a page containing the URL, *sends* this page to the user, and then effectively *suspends* the application waiting for a user interaction via this URL. This interaction is handled by the server extracting and invoking the continuation corresponding to the URL, resuming the computation. Thus, each user interaction corresponds to the invocation of some URL, and therefore the invocation of a continuation.

We will define and implement the desired state mechanism in this context.

Failed Implementation Approaches

We first show, informally, that using the existing mechanisms of the language will *not* work. We then use this to motivate our new solution.

Studying the sequence of interactions, it is unsurprising that a purely functional approach fails to properly capture the desired semantics. Concretely, the state of the component cannot be stored as an immutable lexical binding; if it is, on return from the detour into the administrative section, the sort state reverts to the default. This is shown in Fig. 2, with the error circled in the outcome of action 7. The only alternative is to use store-passing style (SPS). Since this transformation is as invasive as CPS, a transformation that the continuation-based methodology specifically aims to avoid, we do not consider this an effective option.[2] (In Sec. 6.1 we explain why this is a practical, not ideological, concern.)

The most natural form of state is a mutable reference (called a *box* in Scheme), which is tantamount to using an entry in a database or other persistent store. This type of state fails because there is a single box for the entire interaction but, because modifications are not limited to a single subtree, different explorations can interfere with one another (a problem that is manifest in practice on numerous commercial Web sites [18]). Concretely, as Fig. 3 shows, the outcome of actions 6 and 7 would be wrong.

Since the two previous forms of state are not sensitive to continuations, it is tempting to use a form of state that is sensitive to continuations, namely **fluid-let**.[3] An identifier bound by **fluid-let** is bound for the *dynamic extent* of the evaluation of the body of the **fluid-let**. (Recall that this includes the invocation of continuations captured within this dynamic extent.)

[2] It is important to note that the administrative section implementation is just one example of where SPS would be used; SPS, like CPS, is a global transformation that would change our entire program.

[3] PLT Scheme [9] contains a feature called a *parameter*. A parameter is like a fluidly-bound identifier, except that it is also sensitive to threads. However, this distinction is not germane to our discussion, and in fact, parameters fail to satisfy us for the same reasons as **fluid-let**.

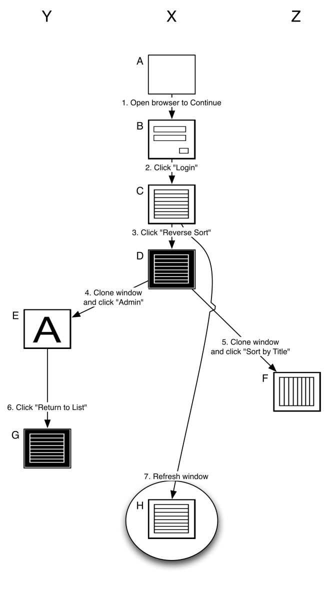
Y
X
Z
A
1. Open browser to Continue
B
2. Click "Login"
C
3. Click "Reverse Sort"
D
4. Clone window and click "Admin"
E
5. Clone window and click "Sort by Title"
F
6. Click "Return to List"
G
7. Refresh window
H

Figure 2. The interaction when *lexical bindings* are used for the sort state without SPS, where the error is circled

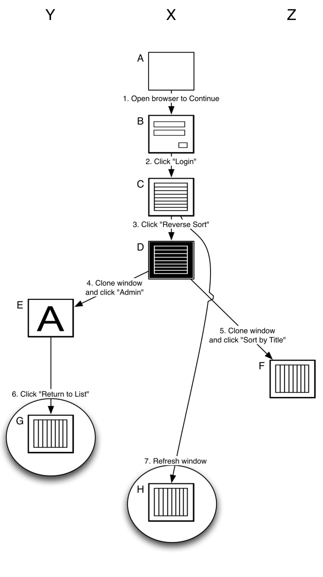
Y
X
Z
A
1. Open browser to Continue
B
2. Click "Login"
C
3. Click "Reverse Sort"
D
4. Clone window and click "Admin"
E
5. Clone window and click "Sort by Title"
F
6. Click "Return to List"
G
7. Refresh window
H

Figure 3. The interaction when a *box* is used for the sort state, where errors are circled

Fluidly-bound identifiers might seem to be a natural way of expressing state in Web applications, because they behave as if they are "closed" over continuations. Since, however, fluidly-bound identifiers are bound in a dynamic extent, any changes to their bindings are lost when the sub-computation wherein they are bound finishes. For instance, a state change in the administrative section would be lost when computation returns out of the extent of that section (here, after action 6). The only alternative is to keep that dynamic extent alive, which would require the entire computation to be written in terms of tail-calls: in other words, by conversion into continuation-passing.

4. Solution

The failed attempts above provide an intuition for why existing approaches to scope and state are insufficient. They fail because the scope in which certain state modifications are visible does not match the scope defined by the semantics of the list component.

4.1 Interaction-Safe State

We define a state mechanism called *Web cells* to meet the state management demands of the list component and similar applications.

An abstraction of Fig. 1 is used to define the semantics of cells. This abstraction is called an *interaction tree*. This tree represents the essential structure of Web interactions and the resolved values of Web cells in the program at each page, and takes into account the subtleties of Refresh, etc. The nodes of this tree are called *frame*s. Frames have one incoming edge. This edge represents the Web interaction that led to the page the frame represents.

Cells are bound in frames. The value of a cell in a frame is defined as either (a) the value of a binding for the cell in the frame; or (b) the value in the frame's parent. This definition allows cell bindings to *shadow* earlier bindings.

Evaluation of cell operations is defined relative to an evaluation context and a frame, called the current frame. The continuations captured by our Web server are designed to close over the current frame—which, in turn, closes over its parent frame, and so on up to the root—in the interaction tree. When a continuation is invoked, it reinstates its current frame (and hence the sequence of frames) so that Web cell lookup obtains the correct values.

Fig. 4 shows the interaction tree after the interactions described by Fig. 1. The actions that modify the sort state create a cell binding in the current frame. For example, when the user logs in to CONTINUE in action 2, the application stores the sort state in frame C with its default value $author$; and, during action 3, $sort$ is bound to $rev(author)$ in frame D. In action 6, the value of $sort$ is $rev(author)$, because this is the binding in frame D, which is the closest frame to G with a binding for $sort$.

The semantics of Web cells is explicitly similar to the semantics of **fluid-let**, except that we have separated the notion of evaluation context and the context of binding. Recall that with **fluid-let**, a dynamic binding is in effect for the evaluation of a specific subexpression. With Web cells, bindings affect evaluations where the current frame is a child of the binding frame.

4.2 Implementation

To implement the above informal semantics, we must describe how frames can be associated with Web interactions in a continuation-based server, so that the relation among frames models the interaction tree accurately. We describe this in the context of our canonical implementation.

Each frame, i.e., node, in the interaction tree is reached by a single action. We regard this single action to be the *creator* of the frame. When the action creates the frame, the frame's parent is the current frame of the action's evaluation. Each invocation of the continuation must create a new frame to ensure the proper behavior with regards to Refresh, as discussed in Sec. 2. Furthermore, each action corresponds to an *invocation* of a continuation. In our implementation, we must ensure that we distinguish between continuation capture and invocation. Therefore, we must change the operation that captures continuations for URLs, **send/suspend**, to create a frame when the continuation is invoked. We will describe how this is done below after we introduce the Web cell primitives.

We summarize the Web cell primitives:

- (**push-frame!**)
  Constructs an empty frame, with the current frame as its parent, and sets the new frame as the current frame.

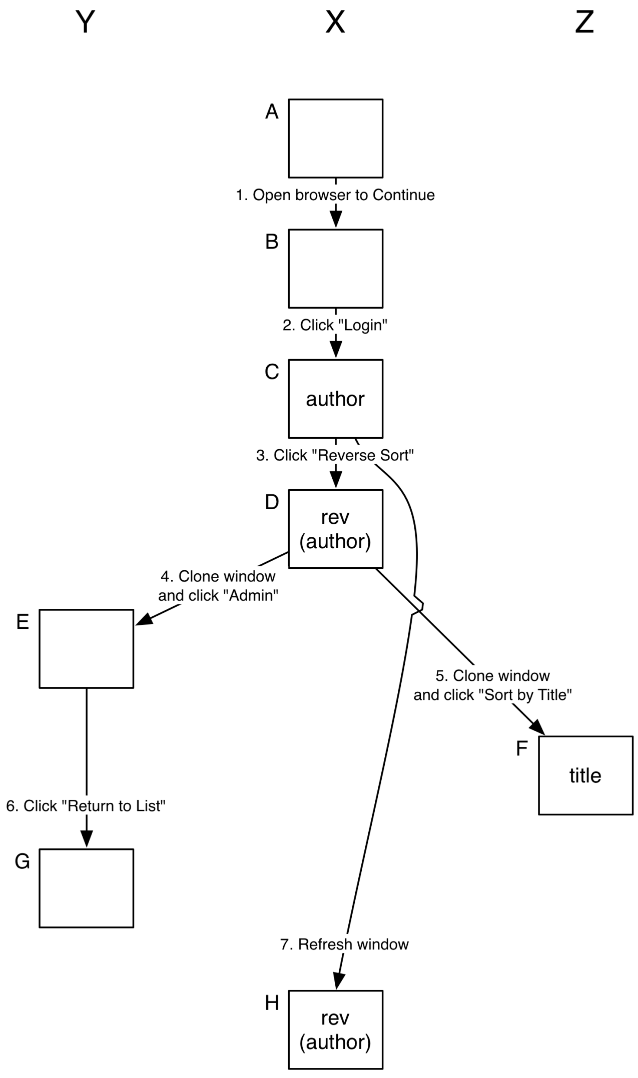
Y
X
Z
A
1. Open browser to Continue
B
2. Click "Login"
C
author
3. Click "Reverse Sort"
D
rev (author)
4. Clone window and click "Admin"
E
5. Clone window and click "Sort by Title"
F
title
6. Click "Return to List"
G
7. Refresh window
H
rev (author)

Figure 4. A diagram of an interaction with CONTINUE, labeled with cells

- (**make-cell** *initial-value*)
  Constructs and returns an opaque Web cell with some initial value, storing it in the current frame.
- (**cell-ref** *cell*)
  Yields the value associated with the cell by locating the nearest enclosing frame that has a binding for the given cell.
- (**cell-shadow** *cell new-value*)
  Creates a binding for the cell in the current frame associating the new value with the cell.

We now re-write **send/suspend** to perform the frame creation accurately. The definition is given in Fig. 5.

To show the other primitives in context, we present an example in Fig. 6. Rather than the list example, which is complicated and requires considerable domain-specific code, we present a simple counter. In this code, the boxed identifier is the interaction-safe Web cell. (The code uses the quasiquote mechanism of Scheme to

```
(define (send/suspend response-generator)
 (begin0
  (let/cc k
   (define k-url (save-continuation! k))
   (define response (response-generator k-url))
   (send response)
   (suspend))
  (push-frame!)))
```

Figure 5. A **send/suspend** that utilizes **push-frame!**

```
(define the-counter (make-cell 0))

(define (counter)
 (define request
  (send/suspend
   (λ (k-url)
    `(html
      (h2 ,(number→string (cell-ref the-counter)))
      (form ((action ,k-url))
       (input ((type "submit") (name "A") (value "Add1")))
       (input ((type "submit") (name "E") (value "Exit"))))))))
 (let ((bindings (request-bindings request)))
  (cond
   ((exists-binding? 'A bindings)
    (cell-shadow
     the-counter
     (add1 (cell-ref the-counter)))
    (counter))
   ((exists-binding? 'E bindings)
    'exit))))

(define (main-page)
 (send/suspend
  (λ (k-url)
   `(html (h2 "Main Page")
          (a ((href ,k-url))
             "View Counter"))))
  (counter)
  (main-page))
```

Figure 6. A Web counter that uses Web cells

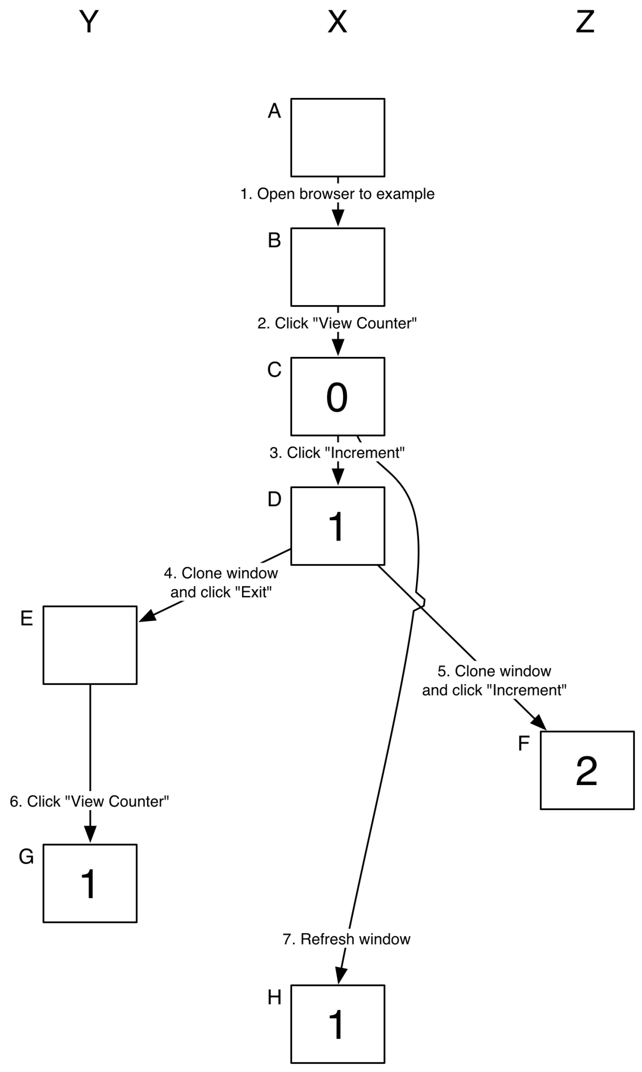

Figure 7. An interaction with the counter (Fig. 6), *structurally identical* to the CONTINUE interaction

represent HTML as an S-expression, and a library function, *exists-binding?*, to check which button the user chose.)

We present interactions with the counter application implemented by the example code in Fig. 6 through the interaction tree diagram in Fig. 7. Like Fig. 1, the links are labeled with the action the user performs. The content of each node represents the value of the counter displayed on the corresponding page. These interactions are specifically chosen to construct a tree that is structurally close to Fig. 1. Therefore, this small example shows the essence of the flow of values in the example from Sec. 2.

The next section formalizes this intuitive presentation of the Web cell primitives.

5. Semantics

The operational semantics, λ_{FS}, is defined by a context-sensitive rewriting system in the spirit of Felleisen and Hieb [7], and is a variant of the λ-calculus with **call/cc** [6] that has been enhanced with terms for representing cells and frames. Evaluation contexts are represented by the nonterminal E and allow evaluations from left-to-right in applications, including in the arguments to the built-in cell manipulation terms.

The semantics makes use of the observation that the only operations on the interaction tree are leaf-to-root lookup and leaf-append. Therefore the semantics, and eventually the implementation, only has to model the current path as a stack of frames. Lookup corresponds to walking this stack, while adding a new node corresponds to pushing a frame onto the stack.

The syntax is given in Fig. 8. The semantics makes use of the domains defined by Fig. 9 for representing stores, frames, and the frame stack. The semantics is defined by the evaluation context grammar and relations in Fig. 10 and the reduction steps in Fig. 11.

The semantics uses short-hand for representing the frame stack in the store. Each frame, ϕ, resides in the store, and $\phi[n \mapsto l]$ represents modification of the store location. The frame stack is represented by the parent pointers in each frame. When a new frame

$$
\begin{aligned}
\cdot\ ;\ \cdot\ ;\ \cdot :: \text{Store} \times \text{Frame Stack} \times \text{Expression} &\longrightarrow \text{Store} \times \text{Frame Stack} \times \text{Expression}\\
\mu\ ;\ \Phi\ ;\ E[((\lambda\ (x_1 \dots x_n)\ e)\ v_1 \dots v_n)] &\longrightarrow \mu\ ;\ \Phi\ ;\ E[e[x_1/v_1, \dots, x_n/v_n]]\\
\mu\ ;\ \Phi\ ;\ E[(\textbf{call/cc}\ e)] &\longrightarrow \mu\ ;\ \Phi\ ;\ E[(e\ (\lambda\ (x)\ (\textbf{abort}\ \Phi\ E[x])))]\\
\mu\ ;\ \Phi\ ;\ E[(\textbf{abort}\ \Phi'\ e)] &\longrightarrow \mu\ ;\ \Phi'\ ;\ e\\
\mu\ ;\ \Phi\ ;\ E[(\textbf{push-frame!})] &\longrightarrow \mu\ ;\ (\emptyset, \Phi)\ ;\ E[(\lambda\ (x)\ x)]\\
\mu\ ;\ (\phi, \Phi)\ ;\ E[(\textbf{make-cell}\ v)] &\longrightarrow \mu[l \mapsto v]\ ;\ (\phi[n \mapsto l], \Phi)\ ;\ E[(\textbf{cell}\ n)]\\
&\ \text{where } n \text{ and } l \text{ are fresh}\\
\mu\ ;\ \Phi\ ;\ E[(\textbf{cell-ref}\ (\textbf{cell}\ n))] &\longrightarrow \mu\ ;\ \Phi\ ;\ E[\ell(\mu, \Phi, n)]\\
\mu\ ;\ (\phi, \Phi)\ ;\ E[(\textbf{cell-shadow}\ (\textbf{cell}\ n)\ v)] &\longrightarrow \mu[l \mapsto v]\ ;\ (\phi[n \mapsto l], \Phi)\ ;\ E[(\textbf{cell}\ n)]\\
&\ \text{where } l \text{ is fresh}
\end{aligned}
$$

Figure 11. The reduction steps of λ_{FS}

$$
\begin{aligned}
v ::=\ & (\lambda\ (x\ \dots)\ e) && (\textit{abstractions})\\
\mid\ & c\\
c ::=\ & (\textbf{cell}\ n) && (\textit{cells})\\
& \text{where } n \text{ is an integer}\\
e ::=\ & v && (\textit{values})\\
\mid\ & x && (\textit{identifiers})\\
\mid\ & (e\ e\ \dots) && (\textit{applications})\\
\mid\ & (\textbf{call/cc}\ e) && (\textit{continuation captures})\\
\mid\ & (\textbf{abort}\ \Phi\ e) && (\textit{program abortion})\\
& \text{where } \Phi \text{ is a frame stack (Fig. 9)}\\
\mid\ & (\textbf{push-frame!}) && (\textit{frame creation})\\
\mid\ & (\textbf{make-cell}\ e) && (\textit{cell creation})\\
\mid\ & (\textbf{cell-ref}\ e\ e) && (\textit{cell reference})\\
\mid\ & (\textbf{cell-shadow}\ e\ e) && (\textit{cell shadowing})
\end{aligned}
$$

Figure 8. Syntax of λ_{FS}

Stores

$$
\begin{aligned}
\mu ::\ & \text{Store}\\
\mu ::=\ & \emptyset && (\textit{empty store})\\
\mid\ & \mu[l \mapsto v] && (\textit{location binding})
\end{aligned}
$$

Frames

$$
\begin{aligned}
\phi ::\ & \text{Frame}\\
\phi ::=\ & \emptyset && (\textit{empty frame})\\
\mid\ & \phi[x \mapsto l] && (\textit{cell identifier binding})
\end{aligned}
$$

Frame Stack

$$
\begin{aligned}
\Phi ::\ & \text{Frame Stack}\\
\Phi ::=\ & \varnothing && (\textit{empty frame stack})\\
\mid\ & \phi, \Phi && (\textit{frame entry})
\end{aligned}
$$

Figure 9. The semantic domains of λ_{FS}

is created, it is placed in the store with its parent as the old frame stack top.

The semantics is relatively simple. The cell and frame operations are quite transparent. We have included **call/cc/frame** (in Fig. 10) as an abbreviation, rather than a reduction step, to keep the semantics uncluttered. If we were to encode it as a reduction step, that step would be:

$$
\begin{aligned}
&\mu\ ;\ \Phi\ ;\ E[(\textbf{call/cc/frame}\ e)] \longrightarrow\\
&\mu\ ;\ \Phi\ ;\ E[e\ (\lambda\ (x)\ (\textbf{abort}\ \Phi\ E[(seqn\ (\textbf{push-frame!})\ x)]))]
\end{aligned}
$$

The order of frame creation in **call/cc/frame** is important. The implementation must ensure that each invocation of a continuation has a unique frame, and therefore a Refresh does not share the same frame as the initial request. The following *faulty* reduction step fails to ensure that each invocation has a unique frame:

$$
\begin{aligned}
&\mu\ ;\ \Phi\ ;\ E[(\textbf{call/cc/frame}\ e)] \longrightarrow\\
&\qquad \mu\ ;\ \Phi\ ;\ E[e\ (\lambda\ (x)\ (\textbf{abort}\ (\text{<new-frame>}, \Phi)\ E[x]))]
\end{aligned}
$$

where <new-frame> is a frame constructed at capture time.

In this erroneous reduction, the new frame is created when the continuation is created, rather than each time it is invoked. Observe that the correct reduction preserves the invariant that each frame has a unique incoming edge in the interaction tree, which this reduction violates.

6. Applications

Web cells answer a pressing need of stateful Web components: they enable (a) defining stateful objects that (b) behave safely in the face of Web interactions while (c) not demanding a strong invariant of global program structure. Other techniques fail one or more of these criteria: most traditional scoping mechanisms fail (b) (as we have discussed in Sec. 4), while store-passing clearly violates (c).

Before we created Web cells, numerous PLT Scheme Web server applications—including ones written by the present authors—used to employ **fluid-let**; based on the analysis described in this paper, we have been able to demonstrate genuine errors in these applications. As a result, PLT Scheme Web server users have adopted Web cells in numerous applications, e.g., a server for managing faculty job applications, a homework turn-in application, a BibTeX front-end, a personal weblog manager, and, of course, CONTINUE itself. We present three more uses of Web cells below.

Semantics

$$eval(e) \text{ holds iff}$$
$$\emptyset \; ; \; (\emptyset, \varnothing) \; ; \; e \longrightarrow^{*} \mu \; ; \; \Phi \; ; \; v$$
$$\text{for some } \mu, \Phi, \text{ and } v$$

Evaluation Contexts

$$\begin{aligned} E \; ::= \; & [] \\ & | \; (v \; \ldots \; E \, e \; \ldots) \\ & | \; (\textbf{make-cell} \; E) \\ & | \; (\textbf{cell-ref} \; E) \\ & | \; (\textbf{cell-shadow} \; E \; e) \\ & | \; (\textbf{cell-shadow} \; v \; E) \end{aligned}$$

Cell Lookup

$$\ell :: \text{Store} \times \text{Frame Stack} \times \text{Location} \rightarrow \text{Value}$$
$$\ell(\mu, (\phi, \Phi), x) \rightarrow v \text{ iff } x \mapsto l \in \phi$$
$$\text{and } l \mapsto v \in \mu$$
$$\ell(\mu, (\phi, \Phi), x) \rightarrow \ell(\mu, \Phi, x) \text{ iff } x \mapsto l \notin \phi$$

Abbreviations

$$(let \; (x \; e_1) \; e_2) \equiv ((\lambda \; (x) \; e_2) \; e_1)$$
$$(seqn \; e_1 \; e_2) \equiv (let \; (x \; c_2) \; c1) \text{ where } x \notin c_2$$
$$(\textbf{call/cc/frame} \; e) \equiv (let \; (c \; e) \; (\textbf{call/cc}$$
$$(\lambda \; (v) \; (seqn \; (\textbf{push-frame!}) \; (c \; v)))))$$

Figure 10. The semantics of λ_{FS}

6.1 Components for Web Applications

Informally, a *component* is an abstraction that can be linked into any application that satisfies the component's published interface. Many of the tasks that Web applications perform—such as data gathering, processing, and presentation—are repetitive and stylized, and can therefore benefit from a library of reusable code.

To maximize the number of applications in which the component can be used, its interface should demand as little as possible about the enclosing context. In particular, a component that demands that the rest of the application be written in store-passing or a similar application-wide pattern is placing an onerous interface on the encapsulating application and will therefore see very little reuse. Stateful components should, therefore, encapsulate their state as much as possible.

We have built numerous Web components, including:

- `list`, whose state is the sort strategy and filter set.
- `table`, which renders a `list` component as a table split across pages, whose state is an instance of the `list` component, the number of list entries to display per page, and the currently displayed page.
- `slideshow`, whose state includes the current screen, the preferred image scale, the file format, etc.

Of the applications described above, every single one had some form of the `list` component, and a majority also had an instance of `table`—all implemented in an ad hoc and buggy manner. Many of these implementations were written using **fluid-let** and did not exhibit the correct behavior. All now use the library component instead.

6.2 Continuation Management

While Web applications should enable users to employ their browser's operations, sometimes an old continuation must expire, especially after completing a transaction. For example, once a user has been billed for the items in a shopping cart, they should not be allowed to use the Back button to change their item selection. Therefore, applications need the ability to manage their continuations.

The PLT Scheme Web server attempts to resolve this necessity by offering an operation that expires all old continuation URLs [14]. This strategy is, however, too aggressive. In the shopping cart example, for instance, only those continuations that refer to non-empty shopping carts need to be revoked: the application can be programmed to create a new shopping cart on adding the first item. In general, applications need greater control over their continuations to express fine-grained, application-specific resource management.

The interaction tree and frame stack associated with each Web continuation provide a useful mechanism to express fine-grained policies. The key feature that is missing from the existing continuation management primitives is the ability to distinguish continuations and selectively destroy them. The current frame stack of a continuation is one useful way to distinguish continuations. Thus, we extend the continuation management interface to accept a predicate on frame stacks. This predicate is used on the frame stack associated with each continuation to decide whether the continuation should be destroyed. For example, in Fig. 4 action 6's continuation could be destroyed based on the Web cell bindings of frame E, such as a hypothetical Web cell storing the identity of the logged-in user.

An application can create a predicate that identifies frames whose destruction corresponds to the above policy regarding shopping carts and purchase. First, the application must create a cell for the shopping cart session. It must then create a new session, A, when the cart goes from empty to non-empty. Then it must remove the session A when the cart becomes empty again and cause continuation destruction if the cart became empty because of a purchase. The predicate will signal destruction for continuations whose frame stack's first binding for the shopping cart session was bound to A. This particular style of continuation management enforces the policy that once a transaction has been committed, it cannot be modified via the Back button.

As another example, consider selective removal of continuations corresponding to non-idempotent requests. These requests are especially problematic in the presence of reload operations, which implicitly occur in some browsers when the user tries to save or print. We can create a cell that labels continuations and a predicate that signals the destruction of those that cannot be safely reloaded. This is a more robust solution to this problem than the Post-Redirect-Get pattern used in Web applications, as we discuss in Sec. 8.3, because it prevents the action from ever being repeated. Thus this infrastructure lets Web application developers give users maximal browsing flexibility while implementing application-specific notions of safety.

6.3 Sessions and Sub-Sessions

A session is a common Web application abstraction. It typically refers to all interactions with an application at a particular computer over a given amount of time starting at the time of login. In the Web cells framework, a session can be defined as the subtree rooted at the frame corresponding to the Logged In page. This definition naturally extends to any number of application-specific sub-session concepts. For example, in CONTINUE it is possible for the administrator to assume the identity of another user. This action logically creates a sub-session of the session started by the initial login action.

The essential code that implements this use case is below:

```
(define-struct session (user))
(define current-session (make-cell #f))
(define (current-user)
  (session-user (cell-ref current-session)))
(define (login-as user)
  (cell-shadow current-session (make-session user)))
(define (add-review paper review-text)
  (associate-review-with-paper
   paper
   (current-user)
   review-text))
```

We now explain each step:

- When the user first logs in, the current-session cell, whose initial value is false, is shadowed by the *login-as* function.
- A new session is created and shadows the old current-session cell, when the administrator assumes the identity of another user via the *login-as* function.
- The *current-user* procedure is called whenever the current user is needed, such as by the *add-review* function. This ensures that the user is tracked by the current session, rather than any local variables.

With this strategy, an administrator can open a new window and assume the identity of a user, while continuing to use their main window for administrative actions. In doing so, the administrator need not worry about leakage of privilege through their identity, since Web cells provide a confinement of that identity in each subtree.

7. Performance

Frames and Web cells leave no significant time footprint. Their primary cost is space. The size of a frame is modest: the smallest frame consumes a mere 266 bytes (on x86 Linux). This number is dwarfed by the size of continuations, of which the smallest is twenty-five times larger. The size of the smallest frame is relevant because it represents the overhead of each frame and the cost to applications that do not use frames. CONTINUE has been used with and without frames in conferences of various sizes without noticeable performance changes in either case.

In practice, however, the space consumed depends entirely on the user's behavior and the structure of the Web application. Two factors are necessary for Web cells to adversely affect memory consumption: (1) users Refreshing URLs numerous times, and (2) the Refreshed pages not allowing further interaction (i.e., not generating additional continuations). We find that in most PLT Scheme Web server applications, most pages enable further interaction, and thus capture additional continuations. As a result, the space for continuations almost always thoroughly dominates that for frames.

8. Related Work

8.1 State and Scope

Recent research has discussed *thread-local storage* in the Java [26] and Scheme [9, 11] communities. In particular, Queinnec [22] deals with this form of scope in the context of a multi-threaded continuation-based Web server. However, in the face of continuations, thread-local store is equivalent to continuation-safe dynamic binders, i.e., parameters [9, 11]. For our purposes, these are the same as **fluid-let**, which we have argued does not solve our problem. Even so, there is much similarity between the semantics of these two types of state.

Web cells and **fluid-let** both install and modify bindings on a stack that represents a node-to-root path in a tree: Frame stacks represent paths in the interaction tree, while **fluid-let** is defined based on program stacks and the dynamic call tree. Operationally, this means that the dynamic call tree is automatically constructed for the programmer (by **fluid-let**) while frame stacks are constructed manually (by **push-frame!**), although the PLT Scheme Web server does this automatically on behalf of the programmer by burying a **push-frame!** inside the implementation of **send/suspend**. Both deal with the complication of interweaving of computation: Web continuations may be invoked any number of times and in any order, while programs with continuations may be written to have a complicated control structure with a similar property. However, to reiterate, **fluid-let** can only help us intuitively understand Web cells, as the two trees are inherently different.

Lee and Friedman [19] introduced quasi-static scope, a new form of scope that has been developed into a system for modular components, such as PLT Scheme Units [8]. This variant of scope is not applicable to our problem, as our composition is over instances of continuation invocation, rather than statically defined (but dynamically composed) software components.

First-class environments [12] are a Lisp extension where the evaluation scope of a program is explicitly controlled by the developer. This work does not allow programs to refer to their own environment in a first-class way; instead, it only allows programs to construct environments and run other programs in them. Therefore, it is not possible to express the way **make-cell** introduces a binding in whatever environment is currently active. Furthermore, this work does not define a semantics of continuations. These limitations are understandable, as first-class environments were created in the context of Symmetric Lisp as a safe way to express parallel computation. However, it may be interesting to attempt to apply our solution to a framework with environments are first-class and try to understand what extensions of such an environment are necessary to accommodate Web cells.

Olin Shivers presents BDR-scope [24] as a variant of dynamic scope defined over a finite static control-flow graph. BDR-scope differs in a fundamental way from our solution, because λ_{FS} allows a variant of dynamic scope defined over a potentially infinite dynamic control-flow tree. However, it may be possible to use Shivers's scope given an alternative representation of Web applications and an analysis that constructed the static control-flow graph representing the possible dynamic control-flows in a Web application by recognizing that recursive calls in the program represent cycles in the control-flow graph. Thus, although not directly applicable, BDR-scope may inspire future research.

Tolmach's Debugger for Standard ML [30] supports a time-travel debugging mechanism that internally uses continuations of earlier points in program executions. These continuations are captured along with the store at the earlier point in the execution. When the debugger "travels back in time", the store locations are unwound to their earlier values. Similarly, when the debugger "travels back to the future", the store is modified appropriately. The essential difference between this functionality and Web cells is that the debugger unwinds all store locations used by the program without exception, while in our context the programmer determines which values to unroll by specifying them as Web cells.

Most modern databases support nested transactions that limit the scope of effects on the database state until the transactions are committed. Therefore, code that uses a database operates with a constrained view of the database state when transactions are employed. A single Web cell representing the current transaction on the database and a database with entries for each cell models the shadowing behavior of those cells. This modeling is accomplished by creating a new transaction, A, after each new frame is created and shadowing the `current-transaction` cell to A. Cell shadowing is possible by modifying the database state. This modification is considered shadowing, because it is only seen by transaction descended from the current transaction, i.e., frames that are descendents of the current frame. The transactions created in this modeling are never finalized. It may be interesting future work to study the meaning of and conditions for finalization and what features this implies for the Web cells model.

8.2 Web Frameworks

Other Web application frameworks provide similar features to the PLT Scheme Web server, but they often pursue other goals and therefore do not discuss or resolve the problems discussed in this paper.

Ruby on Rails [23] is a Web application framework for Ruby that provides a Model-View-Controller architecture. Rails applications are inherently defined over an Object-Relational mapping to some database. The effect of this design is that all state is shared globally or by some application-specific definition of a 'session'. Therefore, Rails cannot support the state management that Web cells offer, nor can it support many other features provided by continuations.

Seaside [5] is a Smalltalk-based Web development framework with continuation support. Seaside contains a very robust system for employing multiple components on a single page and supports a variant of Web cells by annotating object fields as being "backtrack-able." However, they do not offer a formal, or intuitive, account of the style of state we offer, and therefore do not offer comparable principles of Web application construction.

Furthermore, Seaside has other limitations relative to the PLT Scheme Web server. A single component cannot interrupt the computation to send a page to the user without passing control to another component using the `call` method, thereby precluding modal interfaces (such as alerts and prompts). The continuation URLs are not accessible to the program, inhibiting useful reusable components like an email address verifier [17]. Furthermore, Seaside's request-processing system requires a component to specify all subcomponents it might render ahead of time, decreasing the convenience of modularity.

Many Web frameworks are similar to Ruby on Rails, for example Struts [27], Mason [21] and Zope [28]; or, they pursue different goals than the PLT Scheme Web server and Seaside. MAWL [1], <`bigwig`> [2], and JWIG [4] support validation and program analysis features, such as sub-page caching and form input validation, but do not support the Back button or browser window cloning; and WASH/CGI [29] performs HTML validation, offers Back button support, and has sophisticated form field type checking and inference, but does not discuss the problems of this paper. WASH/CGI use of monadic style allows the use of store-passing style for the expression of the programs discussed in this paper. However, we have specifically tried to avoid the use of SPS, so this solution is not applicable to our context.

Java servlets [25] are an incremental improvement on CGI scripts that generally perform better. They provide a session model of Web applications and do not provide a mechanism for representing state with environment semantics, which precludes the representation of Web cells. Thus they do not offer a solution to the problem discussed in this paper.

8.3 Continuation Management

In Sec. 6.2, we discussed how Web cells can be used to organize continuation management as a solution to the problem of selectively disabling old URLs. A sub-problem of this has been addressed by work targeted at preventing the duplication of non-idempotent requests.

The Post-Redirect-Get pattern [10] is one strategy that is commonly used in many different development environments.[4] With this pattern, URLs that represent non-idempotent requests correspond to actions that generate an HTTP Redirect response, rather than an HTML page. This response redirects the browser to an idempotent URL. This strategy exploits the peculiar behavior of many browsers whereby URLs that correspond to Redirect responses are not installed in the History, and are therefore not available via the Back button. However, nothing prevents these URLs from being exposed via network dumps, network latency, or alternative browsers. In fact, this does occur in many commercial applications, forcing developers to employ a combination of HTML and JAVASCRIPT to avoid errors associated with network latency. Therefore, a continuation management strategy that can actually disable non-idempotent URLs provides a more robust, linguistic solution.

Another solution [22] to this problem relies on one-shot continuations [3]. These continuations detect when they are invoked a second time and produce a suitable error. This is easily expressed by the following abstraction:

```
(define send/suspend/once
 (λ (response-generator)
  (define called? (box #f))
  (define result (send/suspend response-generator))
  (if (unbox called?)
     (error 'send/suspend/once "Multiple invocations.")
     (begin (set-box! called? #t)
           result))))
```

However, this strategy cannot be used to implement the shopping cart example without severe transformation of the source program to propagate the *called?* binding to each code fragment that binds URLs. In contrast, our solution requires no transformations of the source, nor does it require any features of Web cells in addition to those presented.

9. Conclusion

We have demonstrated that the connection between continuations and Web computation in the presence of state is subtler than previous research suggests. In particular, a naïve approach inhibits the creation of applications with desirable interactive behavior. Our work explains the problem and provides a solution. We have implemented this solution and deployed it in several applications that are in extensive daily use.

Our result offers several directions for future work. First, we would like to construct an analysis to avoid the cost of unused frames in our implementation, similar to tail-call optimization, which avoids the cost of redundant stack frames. Second, we would like to extend our existing model checker [20] to be able to handle the subtleties introduced by this type of state management. Third, we would like to use the semantics to formally compare the expressive power of Web cells with the other primitives we have discussed in the paper. It appears that we can provide a typed account of Web cells by exploiting those for mutable references, but we have not confirmed this. Finally, we can presumably recast the result in this paper as a monad of the appropriate form.

Acknowledgments

The authors thank Matthew Flatt for his superlative work on MzScheme. Numerous people have provided invaluable feedback

[4] The name of this pattern refers to the HTTP semantics that POST requests are non-idempotent, while GET requests are idempotent. Applications such as proxy servers assume this semantics to safely cache GET requests. However, few Web applications guarantee that GET requests are, in fact, idempotent.

on this work, including Ryan Culpepper, Matthias Felleisen and Dave Herman. Several anonymous reviewers offered numerous invaluable comments on the content and presentation. We also thank the many users of the PLT Scheme Web server and the CONTINUE conference manager. This work was supported by NSF grants CCR-0305949, CPA-0429492, and CCF-0447509.

References

[1] D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Conference on Domain-Specific Languages*, 1997.

[2] C. Brabrand, A. Møller, and M. I. Schwartzbach. The <bigwig> project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.

[3] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–107, 1996.

[4] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Language Systems*, 25(6):814–875, 2003.

[5] S. Ducasse, A. Lienhard, and L. Renggli. Seaside - a multiple control flow web application framework. In *European Smalltalk User Group - Research Track*, 2004.

[6] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[7] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[8] M. Flatt and M. Felleisen. Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[9] M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (or, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, Sept. 1999.

[10] A. J. Flavell. Redirect in response to POST transaction, 2000. `http://ppewww.ph.gla.ac.uk/%7Eflavell/www/post-redirect.html`.

[11] M. Gasbichler, E. Knauel, M. Sperber, and R. A. Kelsey. How to add threads to a sequential language without getting tangled up. In *Scheme Workshop*, Oct. 2003.

[12] D. Gelernter, S. Jagannathan, and T. London. Environments as first class objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–110, 1987.

[13] P. Graham. Beating the averages, Apr. 2001. `http://www.paulgraham.com/avg.html`.

[14] P. T. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. In *European Symposium on Programming*, pages 122–136, Apr. 2001.

[15] P. W. Hopkins. Enabling complex UI in Web applications with send/suspend/dispatch. In *Scheme Workshop*, 2003.

[16] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

[17] S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, pages 2–16, January 2003.

[18] S. Krishnamurthi, R. B. Findler, P. Graunke, and M. Felleisen. Modeling Web interactions and errors. In D. Goldin, S. Smolka, and P. Wegner, editors, *Interactive Computation: The New Paradigm*, Springer Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.

[19] S.-D. Lee and D. P. Friedman. Quasi-static scoping: sharing variable bindings across multiple lexical scopes. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 479–492, New York, NY, USA, 1993. ACM Press.

[20] D. R. Licata and S. Krishnamurthi. Verifying interactive Web programs. In *IEEE International Symposium on Automated Software Engineering*, pages 164–173, Sept. 2004.

[21] Mason HQ. *The Mason Manual*, 2005.

[22] C. Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, pages 23–33, 2000.

[23] Ruby on Rails. *The Ruby on Rails Documentation*, 2005.

[24] O. Shivers. The anatomy of a loop: a story of scope and control. In *ACM SIGPLAN International Conference on Functional Programming*, pages 2–14, 2005.

[25] Sun Microsystems, Inc. *JSR154 - Java^{TM}Servlet 2.4 Specification*, 2003.

[26] Sun Microsystems, Inc. *The Class:ThreadLocal Documentation*, 2005.

[27] The Apache Struts Project. *The Struts User's Guide*. The Apache Software Foundation, 2005.

[28] The Zope Community. *The Zope Book*, 2005.

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

[30] A. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, Apr. 1995.