Modeling Web Interactions and Errors*

Shriram Krishnamurthi[1],
Robert Bruce Findler[2],
Paul Graunke[3,**], and
Matthias Felleisen[3]

[1] Brown University, Providence, RI, USA
[2] University of Chicago, Chicago, IL, USA
[3] Northeastern University, Boston, MA, USA

Summary. Programmers confront a minefield when they design interactive Web programs. Web interactions take place via Web browsers. Browsers permit consumers to whimsically navigate among the various stages of a dialog, leading to unexpected outcomes. Furthermore, the growing diversity of browsers means the number of interactive operations users can perform continues to grow.

To investigate this programming problem, we develop a foundational model of Web interactions that reduces the panoply of browser-supported user interactions to three fundamental ones. We use the model to formally describe two classes of errors in Web programs. The descriptions suggest techniques for detecting both classes of errors. For one class we present an incrementally-checked record type system, which effectively eliminates these errors. For the other class, we introduce a dynamic safety check that employs program annotations to detect errors.

1 Introduction

Over the past decade, the Web has evolved from a static medium into an interactive one. A representative article claims that more than half of all Web transactions are interactive [4], and this ratio only grows in favor of interactivity. Indeed, entire corporations (including book retailers, auction sites, travel reservation services, and so on) now interact primarily or solely through the Web. These interfaces no longer present static content but rather consume user input, perform computation based on these inputs, and generate corresponding output. As a result, the Web has been transformed into an important (and increasingly dominant) medium of interactive computation.

This rapid growth in the volume of interactively generated content might suggest that Web page developers and programmers have mastered the mechanics of interactive Web content. In practice, however, as this paper demonstrates, consumers still

* This research is partially supported by NSF grants CCR-0305949, ESI-0010064 and CAI-0086264.

** Current affiliation: Galois Connections, Inc.

encounter many, and sometimes costly, program errors as they utilize these new services. Furthermore, many of these errors are caused precisely when users employ the *interactive* operations supported by Web browsers. A strong foundation for interactive computation must therefore study and address the world of Web programs.

A Web program's execution consists of a series of interactions between a Web browser and a Web server. When a Web browser submits a request whose path points to a Web program, the server invokes the program with the request via any of a number of protocols (CGI [19], Java servlets [7], or Microsoft's ASP.NET [18]). It then waits for the program to terminate and turns the program's output into a response that the browser can display. Put differently, each individual Web program simply consumes an HTTP request and produces a Web page in response. It is therefore appropriate to call such programs "scripts" considering that they only read some inputs and write some output. This very simplicity, however, is also what makes the design of multi-stage Web dialogs difficult.

First, multi-stage interactive Web programs consist of many scripts, each handling one request. These scripts communicate with each other via external media, because the participants in a dialog must remember earlier parts of a conversation. Not surprisingly, forcing the scripts to communicate this way causes many problems, considering that such communications rely on unstated, and therefore easily violated, invariants.

Second, the use of a Web browser for the consumer's side of the dialog introduces even more complications. The primary purpose of a Web browser is to empower consumers to navigate among a web of hyperlinked nodes at will. A consumer naturally wants this same power to explore dialogs on the Web. For example, a consumer may wish to backtrack to an earlier stage in a dialog, clone a page with choices and explore different possibilities in parallel, bookmark an interaction and come back to it later, and so on. Hence, a programmer must be extremely careful about the invariants that govern the communications among the scripts that make up an interactive Web program. What appears to be invariant in a purely sequential dialog context may not be so in a dialog medium that allows arbitrary navigation actions.

In this paper, we make three contributions to the problem of designing reliable interactive Web programs. First, we develop a simple, formal model of Web interactions. Using this model, we can explain the above problems concisely. Second, we develop a type system that solves one of these problems in a provable manner (relative to the model). Third, because not all the checks can be performed statically, we suggest run-time checks to supplement the type system.

Section 2 describes a problem on an actual corporate Web site that succintly demonstrates the style of problems we study. Section 4 introduces a model of Web interactions suitable for understanding problems with sequential programs. Section 5 uses the model to demonstrate two major classes of mistakes. Section 6 introduces a standard type system for the Web that eliminates the first class of mistakes. Section 7 introduces a dynamic check into the programming language that warns consumers of potential problems. Sections 3 and 8 place our work in context.

2 A Sample Problem

We illustrate one of the Web programming problems with an example from the commercial world. Figure 1 contains snapshots from an actual interaction with Orbitz,[4] which sells travel services from many vendors. It naturally invites comparison shopping. In particular, a customer may enter the origin and destination airports to look for flights between cities, receive a list of flight choices, and then conduct the following actions:

1. Use the "open link in new window" option to study the details of a flight that leaves at 5:50pm (step 1). The consumer now has two browser windows open.
2. Switching back to the choices window (step 2), the consumer can inspect a different option, e.g., a flight leaving at 9:30am (step 3). Now the consumer can perform a side-by-side comparison of the options in two browser windows.
3. After comparing the flight details, the customer decides to take the first flight after all. The consumer switches back to the window with the 5:50pm flight (step 4). Using this window (form), the consumer submits the request for the 5:50pm flight (step 5).

At this point, the consumer expects the reservation system to respond with a page confirming the 5:50pm flight. Alarmingly, even though the page indicates that clicking would reserve on the 5:50pm flight, Orbitz instead selects the 9:30am flight. A customer who doesn't pay close attention may purchase a ticket on the wrong flight.

The Orbitz problem dramatically illustrates our case. Sadly, this is not an isolated error. It exists in other services (such as hotel reservations) on the Orbitz site. Furthermore, as plain consumers, we have stumbled across this and related problems while using several vendor's sites, including Apple, Continental Airlines, Hertz car rentals, Microsoft, and Register.com. Clearly, an error that occurs repeatedly across organizations suggests not a one-time programming fault but rather a systemic problem. Hence, we must develop a foundational model to study Web interactions.

3 Prior Work

The Bigwig project [2] (a descendant of Bell Lab's Mawl project [1]) provides a radical solution to the problem. The main purpose of the project is to provide a domain-specific language for composing interactive Web sessions. The language's runtime system enforces the (informal) model of a session as a pair of communicating threads [3]. For example, clicking on the back button takes the consumer back to the very beginning of the dialog. While such a runtime system prevents damage, it is also overly draconian, especially when compared to other approaches to dealing with Web dialogs.

John Hughes [15], Christian Queinnec [22], and Paul Graham [13] independently had the deep insight that a browser's navigation actions correspond to the use of first-class continuations in a program. In particular, they show that an interaction with the

[4] The screenshots were produced on June 28, 2002.

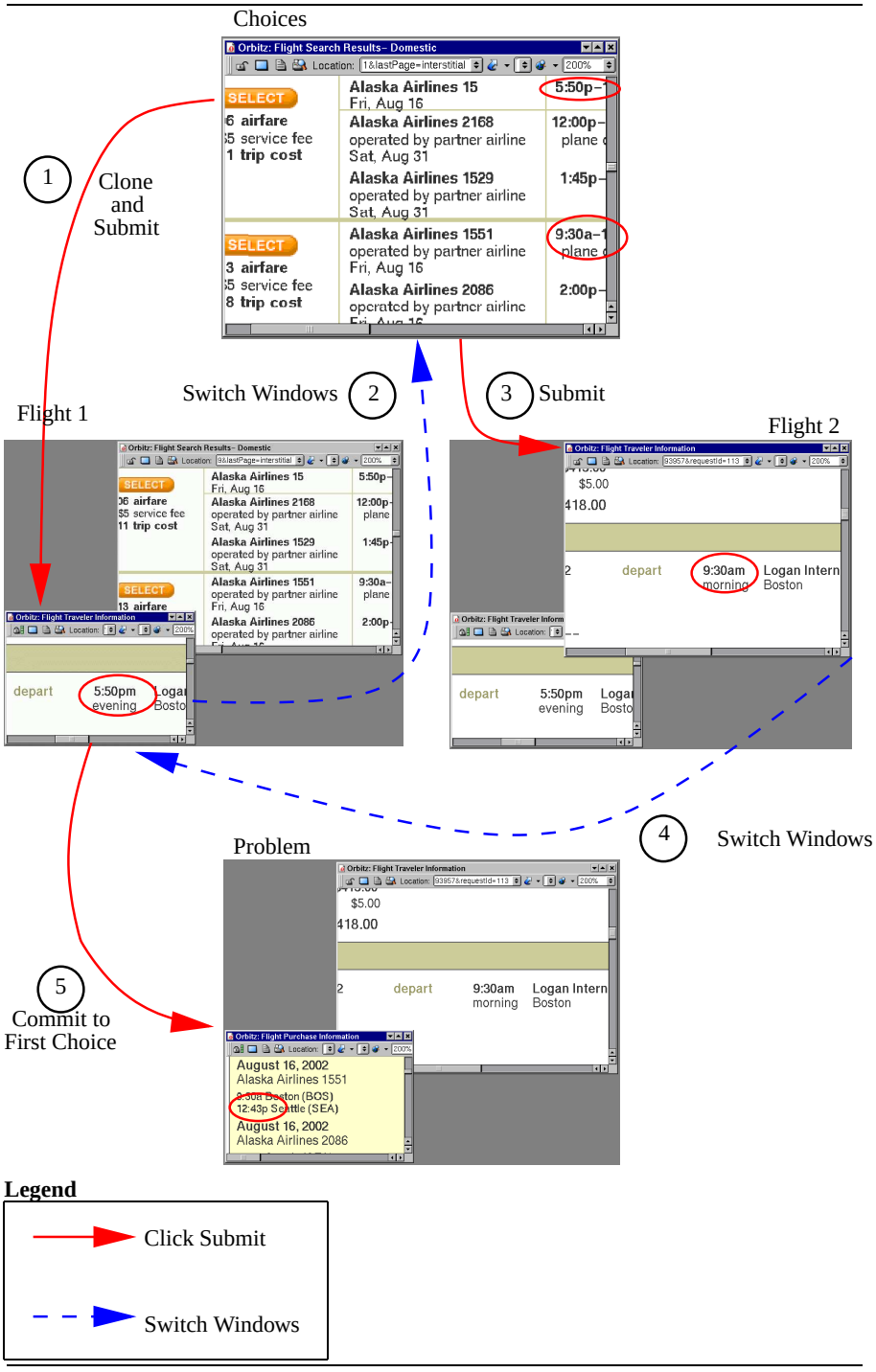

Fig. 1. Orbitz Interactions

consumer corresponds to the manipulation of a continuation. If the underlying language and server support these manipulations, a program doesn't have to terminate to interact with a consumer but instead captures a continuation and suspends the evaluation. Every time a consumer submits a response, the computation resumes the proper continuation. Put differently, the communication among scripts is now internalized within one program and can thus be subjected to the safety checks of the language.

Our prior work explored the implications of Queinnec's in two ways. First, we built a Web server that enables Web programs to interact directly with consumers [14]. Programming in this world eliminates many of the problems in a natural manner. Second, because this solution only applies if the server offers support for storing continuations, we explored the automatic generation of robust Web programs via functional compilation techniques [17]. While this idea works in principle, a full-fledged implementation requires a re-engineered library system and runtime environment for the targeted language.

Thiemann [26] started with Hughes's ideas and provides a monad-based library for constructing Web dialogs. In principle, his solution corresponds to our second approach; his monads take care of the "compilation" of Web scripts into a suitable continuation form. Working with Haskell, Thiemann can now use Haskell's type system to check the natural communication invariants between the various portions of a Web program. This work must accommodate effects (interactions with file systems, data bases, etc.), which it does in a somewhat unnatural manner. Specifically, for each interaction, the CGI scripts are re-executed from the beginning to the current point of interaction, which can be computationally expensive. This monad-based approach does, however, avoid the re-execution of effects, thereby preserving observed behavior relative to these effects.

4 Modeling the Web

As Web browsers proliferate, we expect that both the number and the nature of problems induced by interaction will grow. Browsers are likely to introduce interaction features that are especially convenient to a user but are equally unanticipated by the application developer. It becomes increasingly difficult to reason about the behavior of a program in the context of each particular browser; we would, therefore, benefit from a foundational model that encapsulates a wide variety of these interactions in a small set of primitives, akin to what Turing machines or lambda calculi do for standard computation. This section presents our first attempt at constructing such a model.

The model we present has four characteristics. First, it consists of a single server and a single client, because we wish to study the problems of *sequential* Web interactions. Second, it deals exclusively with dynamically generated Web pages, called forms, to mirror HTML's sub-language of requests. Third, the model allows the consumer to switch among Web pages arbitrarily; as we show later, this suffices to represent the problem in Section 2 and similar phenomena. Finally, the model is abstract

with respect to the programming language so that we can experiment with alternatives; here we use a lambda calculus for forms and basic data, though we could also have used a model such as Classic Java [10].

Our model lacks several properties that are orthogonal to our goals. First, the model ignores client-side storage, a.k.a. "cookies," which primarily addresses customization and storage optimizations. Server-side storage suffices for our goals. Second, Web programmers must address concurrency via locking, possibly relying on a server that serializes each session's requests or relying on a database. Distributing the server software across multiple machines complicates concurrency further. Third, monitoring and restarting servers improves fault tolerance. Fourth, the model does not allow the user to add fields to or drop fields from Web forms before submission. While the HTTP protocol permits this, browsers typically ensure that this does not happen. Accordingly, Web applications can protect themselves against dropped fields through a simple dynamic check that will not, in practice, ever fail. Finally, the model neither addresses nor introduces any security concerns, but existing solutions for ensuring authentication and privacy apply [8, 11].

4.1 Server and Client

Figure 2 describes the components of our model. Each Web configuration (*W*) consists of a single server (*S*) and a single client (*C*). The server consists of storage (Σ) and a dispatcher (see Figure 4). The dispatcher contains a table *P* (for "programs") that associates URLs with programs and an evaluator that applies programs from the table to the submitted form. Programs are closed terms (M°) in a yet to be specified programming language.

$W = S \times C$	{ "", "x", "why", "zee" }	$\subset$ *String*
$S = \Sigma \times P$	{ *x*, *y*, *z* }	$\subset$ *Id*
$P = Url \mapsto M^\circ$	{ *www.drscheme.org*, *www.plt-scheme.org* }	$\subset$ *Url*
$M^\circ =$ *programs*		
$C = F \times \overrightarrow{F}$		
$F = (\textbf{form}\ Url\ \overrightarrow{(Id\ V_\flat)})$		
$V_\flat =$ *Int* | *String*		

Fig. 2. Components of the Web Model

The client consists of the current Web form and a set of all visited Web forms. Initially, the set is a singleton consisting of only the home page. It then grows as the consumer visits additional pages. The model assumes that the consumer can freely (non-deterministically) replace the current page with some previously visited page, or visit a new page. Since the current page is always an element of all previously visited pages, the consumer can also return to this page. We claim that this model of

a consumer represents most interesting browser navigation actions, including some not yet conceived by browser implementors.[5]

The model distills a Web page to a minimal representation. Every page is simply a form (F). It contains the URL to which the form is submitted and a set of form fields. A field names a value that the consumer may edit at will. Figure 3 presents a concrete WebL form and its equivalent in HTML.

```
(form www.plt-scheme.org/my-program.ss
  (name "Paul") (time "1:30"))

<html>
 <body>
  <form action="www.plt-scheme.org/my-program.ss"
        method="post">
    <input type="text" name="name" value="Paul" />
    <input type="text" name="time" value="1:30" />
    <input type="submit" value="Submit">
  </form>
 </body>
</html>
```

Fig. 3. WebL Form and Equivalent HTML Form

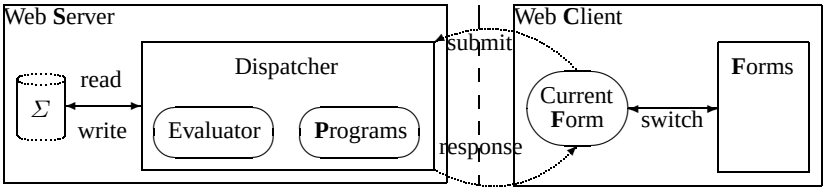

Fig. 4. The Web Picture

Figure 4 illustrates how the pieces of the model interact. The bold-faced letters correspond to the non-terminals in Figure 2. The server and client may run on different machines, connected by a network. The client sends its current form to the server. The form names a program on the server; the server applies this program to the form and produces a response, possibly accessing the store in the process. Finally, the response replaces the current form on the client and appears in the client's set of visited forms.

[5] Entering arbitrary URLs into the browser is a degenerate case of the user creating a brand new form, possibly with an incorrect number of fields (zero) or the wrong field names.

$d_p : \Sigma \times F \longrightarrow \Sigma \times F$

`fill-form` : $W \longrightarrow W$
$\langle s,\ \langle (\textbf{form}\ u\ \overrightarrow{(k\ v_0)}),\ \overrightarrow{f}\rangle\rangle \hookrightarrow \langle s,\ \langle (\textbf{form}\ u\ \overrightarrow{(k\ v_1)}),\ \{(\textbf{form}\ u\ \overrightarrow{(k\ v_1)})\} \cup \overrightarrow{f}\rangle\rangle$

`switch` : $W \longrightarrow W$
$\langle s,\ \langle f_0,\ \overrightarrow{f}\rangle\rangle \hookrightarrow \langle s,\ \langle f_1,\ \overrightarrow{f}\rangle\rangle$ **where** $f_1 \in \overrightarrow{f}$

`submit` : $W \longrightarrow W$
$\langle\langle \sigma_0,\ p\rangle,\ \langle f_0,\ \overrightarrow{f}\rangle\rangle \hookrightarrow \langle\langle \sigma_1,\ p\rangle,\ \langle f_1,\ \{f_1\} \cup \overrightarrow{f}\rangle\rangle$
where $\langle \sigma_1,\ f_1\rangle = d_p(\sigma_0, f_0)$

Fig. 5. Language Transition Relation

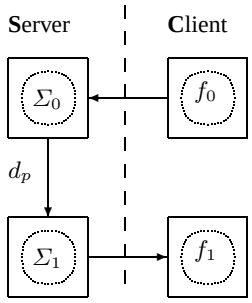

Fig. 6. Client-Server Control Flow

To specify behavior, we use rewriting rules on Web configurations. Figure 5 contains rules that determine the behavior of the client and server as far as Web programs are concerned. Each rule is indexed by an operation and takes a server-client pair to a new server-client pair, reflecting the change caused by the operation.

fill-form allows the client to edit the values of fields in the current form. The form with the new data both becomes the current form and is added to the cache. This rule does not affect the server.

switch brings to the foreground a (possibly) different Web form from the client's repository of visited forms. In practice, this happens in a number of ways: switching active browser windows, revisiting a cached page[6] using the back or forward buttons, or selecting a bookmark. This, too, does not affect the server.

submit dispatches on the current form's URL to find a program in the table *P*. This program consumes the current server state and the submitted form to generate an updated server state and a response form. The server records this new state, while

[6] The actual behavior of revisiting a page depends on whether the page is cached or not. Returning to a non-cached page falls under the *submit* rule.

the new form is sent to the client and becomes the new current form. Figure 6 depicts this flow of control.

The actual dispatching and evaluation (which is triggered by dispatching) are specific to the programming language, which we introduce next.

4.2 Functional Web Programming

Figure 7 specifies WebL, a core Web programming language. WebL extends the call-by-value λ-calculus [21] with integers, strings, and Web forms, which are records with a reference to a program. The language layer connects to the Web layer of the model (Figure 2) by providing the two missing components: the syntax (*M*) and semantics of program evaluation, and the language-sensitive dispatch function d_p.

The **form** construct creates Web forms. The $M.Id$ construct extracts the value of a form field with the name *Id*. We specify the semantics of WebL with a reduction semantics [9]. There are two reductions: the β_v reduction substitutes an argument value for the formal parameter in the body of a function at an application, while the select reduction performs field lookup.

The bottom half of Figure 7 specifies dispatching. It shows how d_p processes a submitted form $form_0$. First, it uses the URL in $form_0$ to extract a program from its table *P*. Second, it applies the program to the form and reduces this application to a value $form_1$. The store σ_0 remains the same, because thusfar WebL has no imperative constructs.

4.3 Stateful Web Programming

Up to this point, scripts in our model can only communicate with each other through forms. In practice, however, Web scripts often communicate not only via forms but also through external storage (such as files and servlet session objects [7]). To model such stateful communications, we extend WebL with **read** and **write** primitives. Figure 8 presents these language extensions. The two primitives empower programs to read flat values from, and to write flat values to, store locations. The reduction relation $\longrightarrow_{v\sigma}$ is the natural extension of the relation $\longrightarrow_v$. The extended relation relates pairs of terms and stores rather than just terms. Consequently the dispatcher starts a reduction with the invoked program and the current store. At the end it uses the modified store to form the next Web configuration. Because only one program may modify the store at a time, the server model is sequential.

5 Problems with Web Applications

Our model of Web interactions can represent some common Web programming problems concisely. Here we present two of them. The first problem is that a Web script expects a different kind of form than is delivered. We dub this problem the "(script) communication problem." The second problem reveals a weakness of the hypertext

Syntax

$$
\begin{array}{rcl}
M & = & V \\
& | & (M\ M) \\
& | & Id \\
& | & (\textbf{form}\ Url\ \overrightarrow{(Id\ M)}) \\
& | & M.Id \\
\\
V & = & V_\flat \mid (\lambda\ (Id)\ M) \mid F
\end{array}
$$

Semantics

$$
\begin{array}{rcl}
E & = & [\,] \mid (E\ M) \mid (V\ E) \\
& | & (\textbf{form}\ Url\ \overrightarrow{(Id\ V)}\ (Id\ E)\ \overrightarrow{(Id\ M)}) \\
& | & E.Id
\end{array}
$$

$$
\begin{array}{ll}
(\beta_v) & E[((\lambda\ (x)\ body)\ v)] \longrightarrow_v E[body[x \backslash v]] \\
(\mathsf{select}) & E[(\textbf{form}\ url\ \overrightarrow{(n_i\ v_i)}\ (n_j\ v_j)\ \overrightarrow{(n_k\ v_k)})\ .\ n_j] \longrightarrow_v E[v_i]
\end{array}
$$

Language to Web Connection

$d_p(\sigma_0, (\textbf{form}\ Url\ \overrightarrow{(Id\ v)})) = \langle \sigma_0,\ form_1 \rangle$
where $prog = P(Url)$ **and** $(prog\ (\textbf{form}\ Url\ \overrightarrow{(Id\ v)})) \longrightarrow_v^* form_1$

Fig. 7. Web Programming Language

Syntax

$M = \cdots \mid (\textbf{read}\ Id) \mid (\textbf{write}\ Id\ M)$

Semantics

$\langle \sigma,\ e_0 \rangle \longrightarrow_{v\sigma} \langle \sigma,\ e_1 \rangle$ **if** $e_0 \longrightarrow_v e_1$
$\langle \sigma,\ E[(\mathbf{write}\ Id\ v_\flat)] \rangle \longrightarrow_{v\sigma} \langle \sigma[Id \backslash v_\flat],\ E[v_\flat] \rangle$
$\langle \sigma,\ E[(\mathbf{read}\ Id)] \rangle \longrightarrow_{v\sigma} \langle \sigma,\ E[\sigma(Id)] \rangle$
where $Id \in dom(\sigma)$, $v_\flat \in V_\flat$

Language to Web Connection

$\Sigma \sqsubseteq (Id \longrightarrow V_\flat)$

$d_p(\sigma_0, (\textbf{form}\ Url\ \overrightarrow{(Id\ s)})) = \langle \sigma_1,\ form_1 \rangle$
where $prog = p(Url)$
$\langle \sigma_0,\ (prog\ (\textbf{form}\ Url\ \overrightarrow{(Id\ s)})) \rangle$
$\longrightarrow_{v\sigma}^* \langle \sigma_1,\ form_1 \rangle$

Fig. 8. Language Extensions for Storage

transfer protocol. Due to the lack of an update method, information on client Web pages becomes obsolete over time and, hence, may mislead the consumer. We dub this problem the "(HTTP) observer problem" indicating that the HTTP protocol does not permit a proper implementation of the Observer pattern [12] (which enables dependent observers to be notified of state changes).

5.1 The Communication Problem

Since standard Web programs must terminate to interact with a consumer, non-trivial interactive software consists of many small Web programs. If the software needs to

interact N times with the client, it consists of $N + 1$ scripts, and all scripts must communicate properly with their successors.[7] Worse, since the client can arbitrarily resubmit pages, the programmer cannot assume anything about the scripts' execution sequence.

```
plt-scheme.org/cgi/start.ss ↦
  (λ (x)
    (form plt-scheme.org/cgi/next.ss
      (name "Your Name")))

plt-scheme.org/cgi/next.ss ↦
  (λ (x)
    (form plt-scheme.org/cgi/done.ss
      (confirm-name x.name)
      (confirm-phone x.phone)))
```

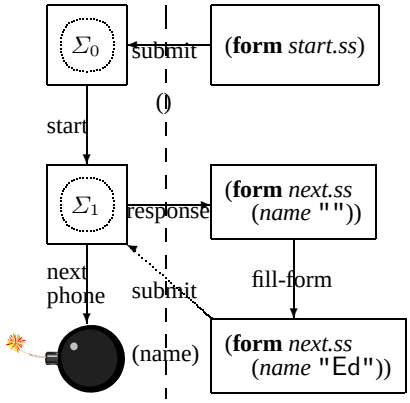

Fig. 9. Collaborating Programs

Even without the difficulties of unusual execution sequences, splitting Web programs into pieces can introduce errors. Consider the example in Figure 9. The server's table contains two programs with the filenames *start.ss* and *next.ss*[8]. The *start.ss* program prompts for the user's name and directs this information to *next.ss*.

[7] A good programmer may recognize opportunities for aggregating some of the programs. It is also possible to use a "multiplexer" technique that merges all these scripts into one single file and uses a dispatcher to find the proper subroutine. The problems remain the same, however, because the various pieces of the same program communicate via HTTP.

[8] Typically, ".ss" is the suffix for Scheme programs; we use it here to be suggestive since our Web programming language is based on Scheme.

This second program attempts to verify some properties about the consumer. In doing so, it assumes that the input form contains both *name* and *phone* fields, and attempts to extract both. The attempt to extract the non-existent *phone* field results in a runtime error. The diagram illustrates the problem graphically. When programmers mistakenly encode field names assumptions into the store—a mistake that is easily made with Java servlet and ASP.NET session objects—these safety errors concerning form field accesses become even more nefarious.

By now, programmers are well-aware of this problem and employ extensive dynamic testing to find these mistakes. In Section 6, we present a type system that discovers such problems statically and still allows programmers to develop complex interactive Web programs in an incremental manner.

5.2 The Observer Problem

In a model-view-controller (MVC) architecture, a change to the model triggers notification to all the views to update their display. Web programs do not enjoy this privilege, because HTTP does not provide for an update (or "push") method. Once a browser receives a page, it becomes outdated when the MVC model changes on the server, which may be due to additional form submissions from the consumer.

The Observer problem is often, but not always, due to a confusion of environments and stores, or form and server-side storage. A program that reserves flights needs to use both kinds of storage to represent different kinds of information [17]. Unfortunately, programmers who don't understand the difference may place information into the store when it really belongs in the Web form.

Figure 10 shows a reformulation of Orbitz's problem (see Section 2) in WebL. The first of these programs, *pick-flight*, asks the customer for a preferred flight time. The second program, *confirm-flight*, writes the selected flight time into external storage before asking the user to confirm the flight time. The third program, *receipt-flight*, reads the selected flight from storage and charges the customer for a ticket.

It is easy to see that the WebL program models the problem in Section 2. Submitting two requests for the *confirm-flight* program results in two pages displaying different flight times on the client, yet only the flight time from the most recent request resides in the server's external storage. Submitting the outdated form that no longer matches the storage produces the mistake.

6 Type Checking Communication

Trying to extract a field from a form fails in WebL if the form does not contain the named field. To prevent such errors, languages often employ a type system (and/or safety checks). Our Web model shows, however, that straightforward type checking doesn't work, because programs consist of many separate scripts loosely connected via forms and storage. Checking all the scripts together is infeasible. Not only are these scripts developed and deployed in an incremental manner, they may also reside

```
pick-flight ↦ (λ (empty-form) (form confirm-flight (departure-time "hh:mm")))

confirm-flight ↦ (λ (first-form)
                   (write your-flight first-form.departure-time)
                   (form receipt-flight (confirm-time (read your-flight))))

receipt-flight ↦ (λ (confirmed-form)
                    (buy-flight (read your-flight))
                    (form next-action (itinerary (read your-flight))))
```

Fig. 10. Stateful Web Programs

on different Web servers and/or be written in different programming languages. Furthermore, consumers can always edit a URL to generate a fresh request that the server has not seen before, akin to a user typing a fresh command at the read-eval-print loop of an interactive language implementation.

We therefore provide an *incremental* type system for Web applications. When the server receives a request for a URL not already in its table, it installs the relevant program to handle the request. Before installing the new program, the server type checks the program, which is a check for internal consistency. In addition, the server also derives constraints that this new program imposes on the other programs on the server with which it interacts. These constraints become external consistency checks. If either type checking or constraint resolution fails, the program is rejected, resulting in an error. In practice, a programmer may register several programs of one application and have them typed checked before they are deployed.

The type system for internal consistency checking heavily borrows from simply-typed λ-calculi with records [5, 20, 24]. Figure 11 defines the type system. In addition to the usual function type ($\longrightarrow$) and primitive types *Int* and *String*, the type language also includes types for Web forms. Similar to record types, **form** types contain the names and types of the form fields that, according to their intended usage, must have flat (marshallable) types. We overload the type environment to map both variables and store locations to types. An initial type environment Γ_0 maps locations in the external storage to flat types. Typed WebL differs from WebL only by requiring types for function arguments. That is, (λ (x) M) becomes (λ (x **:** τ) M) in Typed WebL.

The type system also serves as the basis for external consistency checking. As the type checker traverses the program, it generates constraints on external programs. The type judgments, as shown in Figure 11, have antecedents (above the bar) which, when conjoined, specify a condition. When this condition holds, the consequent (below the bar) also holds. Each judgment rules that a type environment (Γ) proves that a term has a particular type, and generates a (possibly empty) set of constraints. A constraint *Url* **:** (**form** $\overrightarrow{(Id\ \tau_\flat)}$) insists that the program associated with *Url* consume Web forms of type (**form** $\overrightarrow{(Id\ \tau_\flat)}$).

Types

$$\begin{array}{rcl} \textit{Type} & = & \textit{Type} \longrightarrow \textit{Type} \\ & | & (\textbf{form}\ \overrightarrow{(Id\ Type_\flat)}) \\ & | & Type_\flat \\ Type_\flat & = & \textit{String} \mid \textit{Int} \end{array}$$

Type Judgments

$$\Gamma \vdash M : \textit{Type}, \Xi$$

where

$$\Xi = \{Url : (\textbf{form}\ \overrightarrow{(Id\ \tau)})\}$$

Type Derivation Rules

$$\Gamma \vdash \textit{string} : \textit{String}, \{\}$$

$$\Gamma \vdash n : \textit{Int}, \{\}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau, \{\}}$$

$$\frac{\Gamma, x : \tau_x \vdash m : \tau, \xi}{\Gamma \vdash (\lambda\ (x : \tau_x)\ m) : \tau_x \longrightarrow \tau, \xi}$$

$$\frac{\Gamma \vdash m_0 : \tau_x \longrightarrow \tau, \xi_0 \quad \Gamma \vdash m_1 : \tau_x, \xi_1}{\Gamma \vdash (m_0\ m_1) : \tau, \xi_0 \cup \xi_1}$$

$$\frac{\Gamma \vdash m : (\textbf{form}\ \overrightarrow{(Id_a\ \tau_{\flat a})}\ (Id_x\ \tau_{\flat x})\ \overrightarrow{(Id_b\ \tau_{\flat b})}), \xi}{\Gamma \vdash m.Id_x : \tau_{\flat x}, \xi}$$

$$\frac{\overrightarrow{\Gamma \vdash m : \tau_\flat, \xi_m}}{\Gamma \vdash (\textbf{form}\ Url\ \overrightarrow{(Id\ m)}) : (\textbf{form}\ \overrightarrow{(Id\ \tau_\flat)}), \{Url : (\textbf{form}\ \overrightarrow{(Id\ \tau_\flat)})\} \cup \overrightarrow{\xi_m}}$$

$$\frac{\Gamma(l) = \tau_\flat}{\Gamma \vdash (\textbf{read}\ l) : \tau_\flat, \{\}}$$

$$\frac{\Gamma(l) = \tau_\flat \qquad \Gamma \vdash m : \tau_\flat, \xi}{\Gamma \vdash (\textbf{write}\ l\ m) : \tau_\flat, \xi}$$

Fig. 11. Internal Types for WebL

Most type rules in Figure 11 handle constraints in a straightforward manner. Checking atomic expressions yields the empty set of constraints. Checking most expressions that contain subexpressions simply propagates the constraints from checking the subexpressions. The application rule says that if the function position generates constraint ξ_0 and the argument position generates constraint ξ_1, then the entire application expression will generate the union of these, i.e., the constraint $\xi_0 \cup \xi_1$. The only expressions that generate fresh atomic constraints are **form** expressions.

The expression (**form** *Url* $\overrightarrow{(Id\ m)}$) constructs a **form** value, so its type is similar to a record type. This **form** expression also indirectly connects the program associated with *Url* to the form the consumer will submit later. If the type-checker looked up the program associated with *Url* immediately and compared the **form** type with the function's argument type, this would suffice. It would not, however, allow for independent development of connected Web programs. Instead, type checking the **form** expression generates the constraint *Url* **:** (**form** $\overrightarrow{(Id\ \tau_\flat)}$), which must be checked later.

Figure 12 extends the definition of the server state *S* with a set of constraints Ξ. The function *Install-program* adds a new program *m* to the server's table *p* at a given *Url* if the program is internally and externally consistent. That is, the program must type check and the generated constraints must be consistent with the constraints already on the server. A set of constraints is consistent iff the set is a function from URLs to types. The *Constrain* function ensures that the program *m* is well typed, and

Server Extension and Additional Functions

$S = \Sigma \times P \times \Xi$

Install-program **:** *Url* $M\ W \longrightarrow W$

Install-program$(Url, m, \langle\langle \sigma,\ p,\ \xi\rangle,\ c\rangle) = \langle\langle \sigma,\ p[Url \backslash m],$ *Constrain*$(\xi, Url, m)\rangle,\ c\rangle$
when *Consistent*(*Constrain*(*ξ*, *Url*, *m*))

Consistent **:** $\Xi \longrightarrow$ *boolean*
Consistent$(\xi) \equiv$
  (*Url* **:** (**form** $\overrightarrow{(Id_0\ \tau_0)}$)) $\in \xi \wedge$
  (*Url* **:** (**form** $\overrightarrow{(Id_1\ \tau_1)}$)) $\in \xi \Longrightarrow$
  $\overrightarrow{(Id_0\ \tau_0)} = \overrightarrow{(Id_1\ \tau_1)}$

Constrain **:** $\Xi\ Url\ M \longrightarrow \Xi$
Constrain(*ξ*$_0$, *Url*, *m*) $=$
  $\xi_0 \cup \xi_1 \cup \{$*Url* **:** (**form** $\overrightarrow{(Id_{in}\ \tau_{in})}$)$\}$
  where
    $\Gamma_0 \vdash m$ **:** (**form** $\overrightarrow{(Id_{in}\ \tau_{in})}$)
      $\longrightarrow$ (**form** $\overrightarrow{(Id_{out}\ \tau_{out})}$), ξ_1

Fig. 12. Constraint Checking

it extends the existing set of constraints ξ_0 to include constraints generated during type checking ξ_1.

plt-scheme.org/cgi/start.ss $\mapsto$
  (λ ([*x* **:** (**form**)])
    (**form** *plt-scheme.org/cgi/next.ss*
      (*name* "Your Name")))

plt-scheme.org/cgi/next.ss $\mapsto$
  (λ ([*x* **:** (**form** (*name String*) (*phone String*))])
    (**form** *plt-scheme.org/cgi/done.ss*
      (*confirm-name* $x.name$)
      (*confirm-phone* $x.phone$)))

Fig. 13. Typed Collaborating Programs

The incremental type checker catches communication errors, including the one demonstrated in Figure 9. Adding type annotations results in the pair of programs in Figure 13. Type checking produces types and constraints for both programs. The constraints, however, reveal a problem. Checking *start.ss* results in the following constraint:

{ *next.ss* **:** (**form** (*name String*)) }

When the server installs *next.ss*, the *Constrain* function generates this constraint:

{ *next.ss* **:** (**form** (*name String*) (*phone String*)) }

These two constraints are not *Consistent*, so the server rejects the *next.ss* program.

With type annotations, type checking, constraint generation, and constraint checking in place, the system provides three levels of guarantees. The first result shows that individual Web scripts respond to appropriately typed requests without getting stuck.

Proposition 1. *For all m in M, τ in Type, and set of Constraints ξ, if $\Gamma_0 \vdash m : \tau, \xi$ then for some v in V, $m \longrightarrow_v^* v$.*

The proof is essentially the same as the usual proof of strong normalization for the simply-typed lambda calculus.

The second proposition shows that the server does not apply Web programs to forms of the wrong type, as long as the server starts in a good state. Before we can state it, however, we need to explain what it means for a server state to be well-typed and for a submitted form to be well-typed. A server is well-typed when all the programs have function types that map forms to forms and when all the constraints are consistent:

server-typechecks($\langle \sigma,\ p,\ \xi \rangle$) iff *Consistent*($\xi$) and for each *Url* in $dom(p)$,
$\Gamma_0 \vdash p$(*Url*) **:** (**form** $\overrightarrow{(Id_1\ \tau_{\flat 1})}$) $\longrightarrow$ (**form** $\overrightarrow{(Id_2\ \tau_{\flat 2})}$), ξ_{Url} and
$\xi_{Url} \subset \xi$ and *Url* **:** (**form** $\overrightarrow{(Id\ \tau_{\flat 1})}$) $\in \xi$

A form is well typed with respect to a server if it refers to a program on the server that accepts that type of form.

form-typechecks($\langle \sigma,\ p,\ \xi \rangle$, (**form** $Url\ \overrightarrow{(Id\ v_\flat)}$)) iff
there are types $\overrightarrow{\tau_\flat}$ such that $\overrightarrow{\Gamma_0 \vdash v_\flat : \tau_\flat, \{\}}$ and
Url : (**form** $\overrightarrow{(Id\ \tau_\flat)}$) is in ξ and
and $Url \in dom(p)$

Proposition 2. *If server-typechecks(s_0) and form-typechecks(s_0, f_0) then for some* $\langle s_1,\ \langle f_1,\ \overrightarrow{f} \rangle \rangle$,

$$\langle s_0,\ \langle f_0,\ \overrightarrow{f} \rangle \rangle \hookrightarrow_{submit} \langle s_1,\ \langle f_1,\ \overrightarrow{f} \rangle \rangle.$$

If the server's set of constraints is closed, the resulting configuration also guarantees the success of the next submission.

Proposition 3. *If* $\langle \langle \sigma,\ p,\ \xi \rangle,\ \langle f_0,\ \overrightarrow{f} \rangle \rangle \hookrightarrow_{submit} \langle s_1,\ \langle f_1,\ \overrightarrow{f} \rangle \rangle$,

server-typecheck($\langle \sigma,\ p,\ \xi \rangle$), form-typechecks($\langle \sigma,\ p,\ \xi \rangle$, f_0),

and for each constraint Url : (**form** $\overrightarrow{(Id\ \tau)}$) *in ξ, if Url is in $dom(p)$ then*

server-typecheck(s_1) and form-typechecks(s_1, f_1).

In practice these checks only need to be performed upon demand. This strategy makes it possible to incrementally install programs that refer to other programs that have not yet been written and that are used only in rare cases, with the caveat that they are only checked when they are installed.

Alternative Web Programming Languages

It is not necessary to instantiate our model with a functional programming language. Instead, we could have used a language such as `<bigwig>`, which is the canonical imperative while-loop language over a basic data type of Web documents [25].

Furthermore, the <bigwig> language already provides an internal type system that derives and checks information about Web documents. Its type system is stronger than ours, allowing programmers to use complex mechanisms for composing Web documents.

The <bigwig> project and our analysis differ with respect to the ultimate goal. First, our primary goal is to accommodate the existing Web browser mechanisms. In contrast, <bigwig>'s runtime system disables the browser's navigation functionality. Second, we wish to accommodate an open world, where scripts in ASP.NET, Perl, or Python can collaborate. Our propositions show how type checks in the language and in the server can accommodate just this kind of openness. The <bigwig> project does not provide a model and therefore does not provide a foundation for investigating Web interactions in general.

Separating constraints on collaborating programs from the type checking of individual programs lends the system flexibility. For Typed WebL programs, the set of forms produced could be computed simply by examining the program's return type. For other languages the local type checking and the constraint generation may be less connected.

Extending our constraint checking to dynamically typed languages requires a type inference system capable of determining the types of all possible forms a program might produce. Though this is not necessary for Typed WebL, we choose to keep the constraint generation separate to emphasize the independence of the constraints from the languages used for individual scripts.

7 Addressing Outdated Observers

Section 5.2 describes the Observer problem, and points out that it is caused by the Web's lack of a "push" method. Some Web sites simulate pushing data by using a "meta" tag in HTML that forces the generated page to refresh its content periodically. A naïve implementation of this technique suffers from obvious scalability problems. More germane to our discussion, however, is that this does not actually implement the desired user interaction.

To understand this, consider the example in section 2. The user opens a new window in step 1 to explore the flight departing at 5:50pm. When the user examines a different flight in step 3, a push implementation would eventually *update* the information in the window for the 5:50pm flight, to maintain its currency with the server's state. While this makes the flight reservation made after step 5 consistent with the information on the window, it means that the user's mental association of the first window with the flight at 5:50pm has been silently invalidated by the update. This error is just as insidious as that in section 2.

A better solution is to modify the server so that it detects when a submitted form does not reflect the server state. Roughly speaking, this corresponds to the execution of a safety check like the one for array indexing or list destructuring. If the "up-to-date" test fails, the server informs the consumer of the situation, which prevents the erroneous computation from causing further damage. Again, in analogy to safety

checks, the server signals an exception and thus informs the consumer at the earliest opportunity that something went wrong. We believe that this approach is general because it is independent of the scripting language. Further, dynamic checking is an appropriate compromise because these kinds of situations depend on dynamic configurations rather than on static properties of the program.

To check on the datedness of a submitted form, the server must perform some additional bookkeeping. Specifically, determining if something is outdated requires a notion of time, and therefore the server must keep track of time [23]. For us, time is the number of processed submissions. The external storage Σ changes so that it maps locations not only to flat values but also to a timestamp for the last write, i.e., $\Sigma \sqsubseteq \mathit{Id} \longrightarrow \mathit{Time} \times V_\flat$ (compare to the signature in figure 8).

In addition, the server maintains a *carrier* set of all storage locations read or written during the execution of a script. When it sends each page to the consumer, the server adds the current time stamp and this set of locations as an extra hidden field on the page.

With this additional bookkeeping, the server can now check whether each request is up-to-date. When a request arrives, the server extracts both the carrier set and the page creation time. If any of the timestamps attached to the locations in the carrier set are out of date, then the submitted form may be inconsistent with the data in the current server store, and the server signals an exception identifying the out-of-date items:

> A *form* with carrier set *CS* and time stamp *T* submitted to a server with current state σ is **out of date** if and only if any of the locations in *CS* have a time stamp in σ that is larger than *T*.

The actual size of the carrier set will vary based not only on the script's function but also on its implementation (i.e., depending on how stateful it is).

Clearly, a naïve use of this test produces many false positives. For example, a script may use and modify the server state to compute a page counter, a set of advertisements, or other information irrelevant to the consumer. If a form is out of date only for "irrelevant" storage locations, the consumer should clearly not receive a warning. We therefore allow programs to specify whether reading or writing a location in the server state is a relevant or irrelevant action from the consumer's perspective. Assuming that language implementors make this change, the Web server can reduce the carrier set that it collects during a script execution and the number of warnings it issues.

8 Conclusion

Our paper introduces a formal model of sequential, interactive Web programs. We use the model to describe classes of errors that occur when consumers interact with programs using the natural capabilities of Web browsers. The analysis pinpoints two classes of problems with scripting languages and servers.

To remedy the situation, languages used for scripting should come with type checkers that compute the shape of expected forms on the input side and the shape of forms that the scripts may produce. These languages should also allow scripts to specify which actions on the server's state are relevant for the consumer. Furthermore, servers should be modified to integrate the type information from the scripts. In particular, servers should only submit forms to a script if the form is well-typed and its content is up-to-date.

Most combinations of Web servers and Web application programming languages fail to implement either kind of test. All of them, in particular, fail to check for the currency of data, even those whose authors are keenly aware of the problem described in Section 2. While we have implemented our model in a toy Web server, we have not (yet) ported the code to our PLT Web server [14]. Similarly, WASH/CGI [26] is based on a purely functional programming language in recognition of the problems involving state; the careful management of state appeares to address the problem of Section 2. This design is, however, deceiving. The true culprit is a lack of server-based checks that warn users about outdated information.

This formal model has already proven useful in other work. Web programs naturally give rise to temporal properties governing their execution over the course of a workflow, making model checking [6] an attractive verification technique. A naïve model construction based purely on the program source, however, fails to take into consideration the many interaction possibilities introduced by browsers, and thus fails to catch errors of the sort discussed in this paper. To model each browser primitive would, however, be onerous. Our work on model checking of Web programs [16] therefore uses the model of this paper to constrain the language of analysis, and can thus verify programs that operate in any browser so long as all their interaction primitives can be reduced to the ones presented in this paper.

In short, the formal model helps us to first reduce the complexity of Web interaction primitives to a small and manageable number. It then helps us describe common Web problems in terms of these primitives. We can then derive verification techniques to address these problems. We hope to exploit this knowledge to build better languages for programming applications that reside on servers and in Web browsers.

Acknowledgment

Thanks to Jacob Matthews for helping us experiment with WASH/CGI, and to Scott Smolka for his careful editorial work.

References

1. Atkins, D. L., T. Ball, G. Bruns and K. C. Cox. Mawl: A domain-specific language for form-based services. *Software Engineering*, 25(3):334–346, 1999.
2. Brabrand, C., A. Møller, A. Sandholm and M. Schwartzbach. A language for developing interactive Web services, 1999. Unpublished manuscript.
3. Brabrand, C., A. Møller, A. Sandholm and M. I. Schwartzbach. A runtime system for interactive Web services. In *Journal of Computer Networks*, pages 1391–1401, 1999.

4. BrightPlanet. DeepWeb. `http://www.completeplanet.com/Tutorials/DeepWeb/`.
5. Cardelli, L. Type systems. In *Handbook of Computer Science and Engineering*. CRC Press, 1996.
6. Clarke, E., O. Grumberg and D. Peled. *Model Checking*. MIT Press, 2000.
7. Coward, D. Java servlet specification version 2.3, October 2000. `http://java.sun.com/products/servlet/`.
8. Dierks, T. and C. Allen. The transport layer security protocol, January 1999. `http://www.ietf.org/rfc/rfc2246.txt`.
9. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992. Original version in: Technical Report 89-100, Rice University, June 1989.
10. Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, January 1998.
11. Freier, A. O., P. Karlton and P. C. Kocher. Secure socket layer 3.0, November 1996. IETF Draft `http://wp.netscape.com/eng/ssl3/ssl-toc.html`.
12. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
13. Graham, P. Beating the averages. `http://www.paulgraham.com/avg.html`.
14. Graunke, P. T., 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, April 2001.
15. Hughes, J. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.
16. Licata, D. R. and S. Krishnamurthi. Verifying interactive Web programs. In *IEEE International Symposium on Automated Software Engineering*, pages 164–173, September 2004.
17. Matthews, J., R. B. Findler, P. T. Graunke, S. Krishnamurthi and M. Felleisen. Automatically restructuring programs for the Web. *Automated Software Engineering: An International Journal*, 2003.
18. Microsoft Corporation. `http://www.microsoft.com/net/`.
19. NCSA. The Common Gateway Interface. `http://hoohoo.ncsa.uiuc.edu/cgi/`.
20. Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.
21. Plotkin, G. D. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.
22. Queinnec, C. 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. Reed, D. P. Implementing atomic actions on decentralized data. In *ACM Transactions on Computer Systems*, pages 234–254, February 1983.
24. Rémy, D. Typechecking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 77–88, 1989.
25. Sandholm, A. and M. I. Schwartzbach. A type system for dynamic Web documents. In *Symposium on Principles of Programming Languages*, pages 290–301, 2000.
26. Thiemann, P. WASH/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Practical Applications of Declarative Languages*, pages 192–208, 2002.