Modeling Web Interactions

Paul Graunke (Northeastern University),
Robert Bruce Findler (University of Chicago),
Shriram Krishnamurthi (Brown University), and
Matthias Felleisen (Northeastern University)

Abstract. Programmers confront a minefield when they design interactive Web programs. Web interactions take place via Web browsers. With browsers, consumers can whimsically navigate among the various stages of a dialog and can thus confuse the most sophisticated corporate Web sites. In turn, Web services can fault in frustrating and inexplicable ways. The quickening transition from Web scripts to Web services lends these problems immediacy.

To address this programming problem, we develop a foundational model of Web interactions and use it to formally describe two classes of errors. The model suggests 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, which catches the mistakes relative to programmers' simple annotations.

1 Introduction

Over the past decade, the Web has become an interactive medium. Far more than half of all Web transactions are interactive [4]. While this rapid growth suggests that Web page developers and programmers have mastered the mechanics of interactive Web content, consumers still encounter many, and sometimes costly, program errors as they utilize these new services. In short, designing interactive Web programs poses interesting and complex problems.

To understand these problems, let us briefly recall how Web programs work. 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 [16], Java servlets [6], or Microsoft's ASP.NET [15]). 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, also 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 oft-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 in a graph 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 communication 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 whimsical navigation actions.

In this paper, we make three contributions to the problem of designing reliable interactive Web programs. First, we develop a simple but 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 runtime checks to supplement the type system.

2   A Sample Problem

Let us illustrate one of the Web programming problems with a commercial example. Figure 1 contains snapshots from an actual interaction with Orbitz,[1] 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 some 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. The consumer now has two browser windows open.
2. Switching back to the choices window, the consumer can inspect a different option, e.g., a flight leaving at 9:30am. 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. Using this window (form), the consumer submits the request for the 5:50pm flight.

At this point, the consumer expects the reservation system to respond with a page confirming the 5:50pm flight. Alarmingly, even though the page says a click on some link would reserve the 5:50pm flight, Orbitz instead chooses the 9:30am flight. A customer who doesn't pay close attention may end up reserving the wrong flight.

The Orbitz problem dramatically illustrates our case. Sadly, this is not an isolated error. Rather 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 suggest not a one-time programming fault but rather a systemic problem. Hence, we believe that it is time to develop a foundational model. Before we do so, however, we review related attempts at overcoming such programming problems.

---

[1] The screenshots were produced on June 28, 2002, but the problems persist as of October 24.
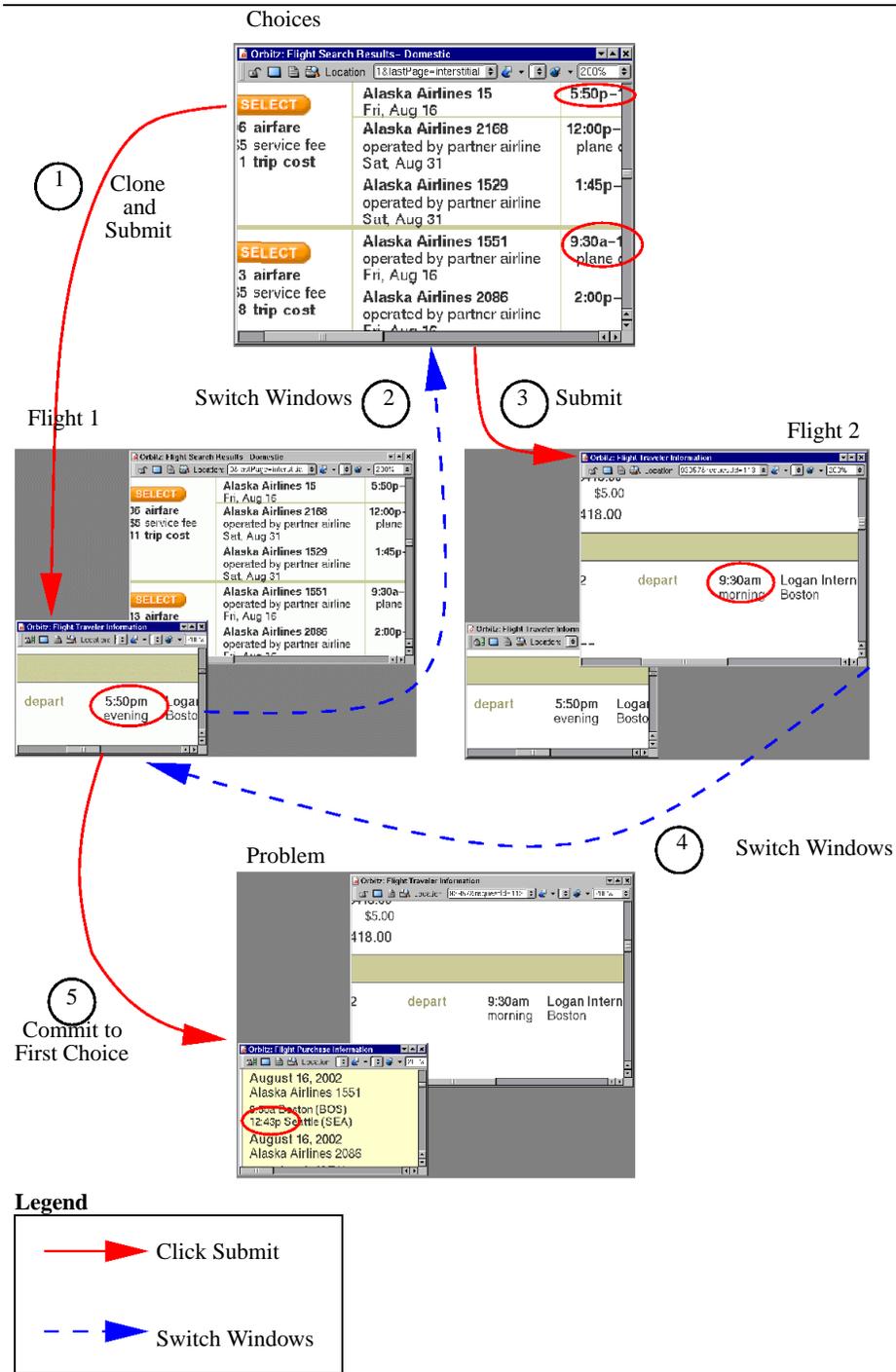
Choices

Fig. 1. Orbitz 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 [14], Christian Queinnec [18], and Paul Graham [11] 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 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 mechanisms 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 [13]. Programming in this world eliminates many of the Web design problems in a natural manner. Second, after we realized that this solution doesn't apply to languages without such mechanisms, we explored the automatic generation of robust Web programs via functional compilation techniques [12]. While this idea works in principle, we recognized that a full-fledged implementation requires a re-engineered library system and runtime environment for the targeted language (say Perl).

Thiemann [21] 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. Haskell, however, is also a problem because Thiemann must accommodate effects (interactions with file systems, data bases, etc) in an unnatural manner. Specifically, for each interaction, his CGI scripts are re-executed from the beginning to the current point of interaction. Even though his monad-based approach avoids the re-execution of effects, it is indicative of the problems with Thiemann's approach. Like our second solution, Thiemann's approach won't easily apply to other languages.

4 Modeling the Web

To study the problems of designing interactive Web programs, we formulate a model with four characteristics. First, it consists of a single server and a single client, because we wish to study the problems of simple sequential Web dialogs. 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 "Orbitz problem" and similar errors. Finally, the model is abstracted over the programming language so that we can experiment with alternatives; here we use a λ calculus for forms and basic data.

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. The model neither addresses nor introduces any security concerns, so existing solutions for ensuring authentication and privacy apply [7, 9].

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 3). The latter contains a table (P) 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 language.

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

Fig. 2. The Web

The client consists of the current Web form and a set of all previously visited Web forms. The set of previously visited forms starts as a singleton set: 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. 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 all interesting browser navigation actions, including those not yet conceived by current browser implementors.

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 illustrates how the pieces of the model interact. The server and client may run on different machines, connected by a network. The client sends its current form
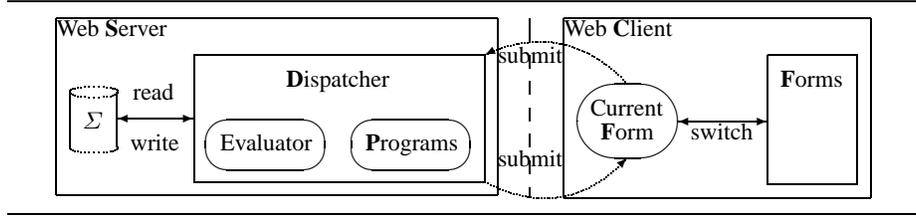
Fig. 3. The Web Picture

to the server. The server applies a program to the form and then 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.

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

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

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

Fig. 4. Transitions

To specify behavior, we use rewriting rules to relate Web configurations. Figure 4 contains rules that determine the behavior of the client and server as far as Web programs are concerned. The *fill-form* rule allows the client to edit the values of fields in the current form. The *switch* rule brings a different Web form to the foreground. In practice, this happens in a number of ways: switching active browser windows, revisiting a cached page[2] using the back or forward buttons, or selecting a bookmark. The *submit* rule dispatches on the current form's URL to run the program found in table *p*, resulting in a new current client form and updated server storage. The actual dispatching and the evaluation are specific to the chosen programming language, which we introduce next.

4.2 Functional Web Programming

Figure 5 specifies the WrForm programming language. WrForm extends the call-by-value λ-calculus with integers, strings, and Web forms, which are records with a reference to a program. The programming language connects to the Web model (figure 2) in three ways: the syntax for forms, the syntax for terms (*M*), and the dispatch function d_p.

---

[2] Returning to a non-cached page falls under the *submit* rule.

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 WrForm with a reduction semantics [8]. There are two reductions: β_v and **select**.

The bottom half of figure 5 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.

Syntax   **Semantics**

$M = V$ $E \quad = [] \mid (E\ M) \mid (V\ E)$
$\quad \mid\ (M\ M)$ $\quad\quad \mid\ (\textbf{form}\ url\ \overrightarrow{(id\ V)}\ (id\ E)\ \overrightarrow{(id\ M)})$
$\quad \mid\ Id$ $\quad\quad \mid\ E.Id$
$\quad \mid\ (\textbf{form}\ Url\ \overrightarrow{(Id\ M)})$
$\quad \mid\ M.Id$ $(\beta_v) \quad E[((\lambda\ (x)\ body)\ v)] \longrightarrow_v E[body[x\backslash v]]$
                                 $(\textsf{select})\ E[(\textbf{form}\ url\ \overrightarrow{(n_i\ v_i)}\ (n_j\ v_j)\ \overrightarrow{(n_k\ v_k)}).\ n_j] \longrightarrow_v E[v_i]$
$V\ = V_\flat \mid (\lambda\ (Id)\ M) \mid \text{F}$

Language to Web Connection

$d_p : \Sigma \times F \longrightarrow \Sigma \times F$
$d_p(\sigma_0, (\textbf{form}\ url\ \overrightarrow{(id\ v)})) = \langle \sigma_0,\ form_1 \rangle$
where $prog = p(url)$ **and** $(prog\ (\textbf{form}\ url\ \overrightarrow{(id\ v)})) \longrightarrow^*_v form_1$

Fig. 5. Web Programming Language

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 (files, session objects). To model such stateful communications, we extend WrForm with **read** and **write** primitives. Figure 6 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. Thus, the server model remains sequential and does not include any concurrency.

5 Problems with the Web

Our model of Web interactions can represent some common Web programming problems concisely. The first problem is that a Web script expects a different kind of form

Syntax	Language to Web Connection

Syntax

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

Semantics

$$\frac{e_0 \longrightarrow_v e_1}{\langle \sigma,\ e_0 \rangle \longrightarrow_{v\sigma} \langle \sigma,\ e_1 \rangle}$$

$\langle \sigma,\ E[(\textbf{write } id\ v)] \rangle \longrightarrow_{v\sigma} \langle \sigma[id\backslash v],\ E[v] \rangle$

$\langle \sigma,\ E[(\textbf{read } id)] \rangle \longrightarrow_{v\sigma} \langle \sigma,\ E[\sigma(id)] \rangle$

where $id \in dom(\sigma)$

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 $prog = p(url)$

$\quad \langle \sigma_0,\ (prog\ (\textbf{form } url\ \overrightarrow{(id\ s)})) \rangle$

$\quad \longrightarrow^*_{v\sigma} \langle \sigma_1,\ form_1 \rangle$

Fig. 6. Language Extensions for Storage

than is delivered. We dub this problem the "(script) communication problem." The second problem reveals a weakness of the hypertext transfer protocol. Due to the lack of an update method, information on client Web pages becomes obsolete and misleads 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 [10].

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.[3] Worse, since the client can arbitrarily resubmit pages, the programmer cannot assume anything about the scripts' execution sequence.

Even without the difficulties of unusual execution sequences, splitting Web programs into pieces can introduce errors. Consider the example in figure 7. The server's table contains two programs: *start.ss* and *next.ss*. The *start.ss* program prompts for the user's name and directs this information to *next.ss*. 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 such 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 the next section, we present a type system that discovers such problems statically and still allows programmers to develop complex interactive Web programs in an incremental manner.

---

[3] 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.
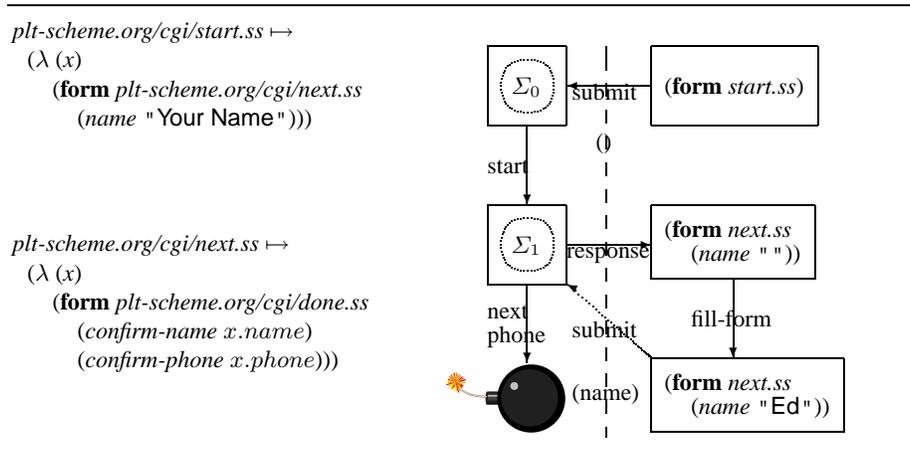
plt-scheme.org/cgi/start.ss $\mapsto$
  (λ (*x*)
    (**form** *plt-scheme.org/cgi/next.ss*
      (*name* "Your Name")))

Σ_0 submit   (**form** *start.ss*)

()

start

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

Σ_1 response   (**form** *next.ss* (*name* ""))

next
phone   submit   fill-form

(name)   (**form** *next.ss* (*name* "Ed"))

Fig. 7. Collaborating Programs

5.2 The Observer Problem

In a model-view-controller (MVC) architecture, each change to the model notifies 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 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. Clearly, a program that reserves flights needs both. Unfortunately, programmers who don't understand the difference may place information into the store when it really belongs in the Web form.

Figure 8 shows a reformulation of Orbitz's problem (see section 2) in WrForm. 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 WrForm 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 one flight time 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 WrForm 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

---

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. 8. Stateful Web Programs

scripts developed and deployed in an incremental manner, they may also reside on different Web servers and/or be written in different programming languages.

We, therefore, provide an incremental type system for Web applications. When the server receives a request for an unknown URL, it installs a program into its table to handle the request. Before installing the new program, the server type checks the program for "internal consistency." In addition, the server also derives constraints that this new program imposes on other Web programs. We refer to this second step as "external consistency" checking. If either step 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, 17, 19]. Figure 9 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, which—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.[4] Typed WrForm differs from WrForm only by requiring types for function arguments. That is, (λ (*x*) *M*) becomes (λ (*x* : *τ*) *M*) in typed WrForm.

The type system also serves as the basis for external consistency checking. As it traverses the program, it generates constraints on external programs. Each type judgment, as shown in figure 9, includes a set of constraints. A constraint *url* **:** (**form** $\overrightarrow{(id\ \tau_\flat)}$) insists that the program associated with *url* consumes Web forms of type (**form** $\overrightarrow{(id\ \tau_\flat)}$).

Most type rules in figure 9 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 only expressions that generate 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

---

[4] The environment Γ_0 is fixed when beginning to check an individual program, but programmers may add extra locations for new programs.

Types

$$Type = Type \longrightarrow Type$$
$$\quad\ \ |\ (\textbf{form}\ \overrightarrow{(Id\ Type_\flat)})$$
$$\quad\ \ |\ Type_\flat$$
$$Type_\flat = String\ |\ Int$$

Type Judgments

$$\Gamma \vdash M : Type, \Xi$$
where
$$\Xi = \{url : (\textbf{form}\ \overrightarrow{(id\ \tau)})\}$$

Type Derivation Rules

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

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

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

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

$$\frac{\begin{array}{c}\Gamma \vdash m_0 : \tau_x \longrightarrow \tau, \xi_0 \\ \Gamma \vdash m_1 : \tau_x, \xi_1\end{array}}{\Gamma \vdash (m_0\ m_1) : \tau, \xi_0 \cup \xi_1}$$

$$\frac{\Gamma \vdash m : (\textbf{form}\ \overrightarrow{(id_a\ \tau_{\flat a})}\ (id_x\ \tau_{\flat x})\ \overrightarrow{(id_b\ \tau_{\flat b})}), \xi}{\Gamma \vdash m.id_x : \tau_{\flat x}, \xi}$$

$$\frac{\overrightarrow{\Gamma \vdash m : \tau_\flat, \xi_m}}{\begin{array}{c}\Gamma \vdash (\textbf{form}\ url\ \overrightarrow{(id\ m)}) : (\textbf{form}\ \overrightarrow{(id\ \tau_\flat)}), \\ \{url : (\textbf{form}\ \overrightarrow{(id\ \tau_\flat)})\} \cup \overrightarrow{\xi_m}\end{array}}$$

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

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

Fig. 9. Internal Types for WrForm

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 10 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 okay. 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.[5] The *Constrain* function ensures that the program *m* is well typed, and it extends the existing set of constraints ξ_0 to include constraints generated during type checking ξ_1.

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

Theorem 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 second theorem 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 the theorem, though, we need to explain what it means for a server state to be well-typed

---

[5] Relaxing this restriction could allow forms to contain extra, unanticipated fields.

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 **:** $\Xi \longrightarrow boolean$	*Constrain* **:** $\Xi\ url\ M \longrightarrow \Xi$
Consistent(ξ) $\equiv$	*Constrain*(ξ_0, *url*, *m*) $=$
$(url : (\textbf{form}\ \overrightarrow{(id_0\ \tau_0)})) \in \xi\ \wedge$	$\xi_0 \cup \xi_1 \cup \{url : (\textbf{form}\ \overrightarrow{(id_{in}\ \tau_{in})})\}$
$(url : (\textbf{form}\ \overrightarrow{(id_1\ \tau_1)})) \in \xi \Longrightarrow$	*where*
$\overrightarrow{(id_0\ \tau_0)} = \overrightarrow{(id_1\ \tau_1)}$	$\Gamma_0 \vdash m : (\textbf{form}\ \overrightarrow{(id_{in}\ \tau_{in})})$
	$\longrightarrow (\textbf{form}\ \overrightarrow{(id_{out}\ \tau_{out})}), \xi_1$

Fig. 10. Constraint Checking

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 $dom(p)$,
$\quad \Gamma_0 \vdash p(url) : (\textbf{form}\ \overrightarrow{(id_1\ \tau_{\flat 1})}) \longrightarrow (\textbf{form}\ \overrightarrow{(id_2\ \tau_{\flat 2})}), \xi_{url}$ and
$\quad \xi_{url} \in \xi$ and $url : (\textbf{form}\ \overrightarrow{(id\ \tau_{\flat 1})}) \in \xi$

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($\langle\sigma,\ p,\ \xi\rangle, (\textbf{form}\ url\ \overrightarrow{(id\ v_\flat)})$) iff
$\quad$ there are types $\overrightarrow{\tau_\flat}$ such that $\overrightarrow{\Gamma_0 \vdash v_\flat : \tau_\flat, \{\}}$ and $url : (\textbf{form}\ \overrightarrow{(id\ \tau_\flat)})$ is in ξ

Theorem 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.

Theorem 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 $url : (\textbf{form}\ \overrightarrow{(id\ \tau)})$ *in* ξ, *url is in* $dom(p)$ *then*
server-typecheck(s_1) and form-typechecks(s_1, f_1).

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 [20]. 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 back button. Second, we wish to

accommodate an open world, where scripts in ASP.NET, Perl, or Python can collaborate. Our theorems 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 WrForm, the set of forms produced could more easily be computed 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.

7 Notifying Outdated Observers

When a script creates a form, it reflects the server's current state. Due to HTTP's shortcomings, a form can lose currency with the server's state. Submitting such a form may, from the consumer's perspective, result in incomprehensible or erroneous behavior.

One way to avoid such errors is to reload pages periodically. Since pages are generated with scripts, reloading implies re-executing scripts. Of course, the re-execution must avoid a duplication of effects on the state of the server, which is precisely what Thiemann's work enables [21]. Unfortunately, this solution doesn't work in general for a number of reasons, some of which were discussed in the section on prior work.[6]

An alternative and general method 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, and that dynamic checking is the appropriate compromise, because these kinds of situations depend on dynamic configurations rather than statically predictable properties.

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. For us, time is the number of processed submissions. The external storage changes so that it maps locations not only to flat values but also to a timestamp for the last **write**: $\Sigma \sqsubseteq Id \longrightarrow Time \times V_\flat$.

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

---

[6] A WASH-CGI program with the problem demonstrated in figure 1, built using WASH-CGI-1.0 downloaded on October 8, 2002, compiled without complaint using GHC-5.02.2 and "reserved" the wrong flight when run. Unfortunately, the program is too long to include in (the margin of) this paper.

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 current server storage:

> 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*.

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 up-to-date.

In short, the formal model helps us to understand what the problems are and which components of the Web should change to avoid such interactions. We have implemented a first prototype of our results and hope to report on experiments with the improved server and servlet language in the near future.

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. Coward, D. Java servlet specification version 2.3, October 2000.
   `http://java.sun.com/products/servlet/`.

7. Dierks, T. and C. Allen. The transport layer security protocol, January 1999. `http://www.ietf.org/rfc/rfc2246.txt`.

8. 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.

9. 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`.

10. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

11. Graham, P. Beating the averages. `http://www.paulgraham.com/avg.html`.

12. Graunke, P., R. B. Findler, S. Krishnamurthi and M. Felleisen. Automatically restructuring programs for the Web. In *IEEE International Conference on Automated Software Engineering*, pages 211–222, 2001.

13. Graunke, P., 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, 2001.

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

15. Microsoft Corporation. `http://www.microsoft.com/net/`.

16. NCSA. The Common Gateway Interface. `http://hoohoo.ncsa.uiuc.edu/cgi/`.

17. Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.

18. 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.

19. Rémy, D. Typechecking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages*, pages 77–88, 1989.

20. Sandholm, A. and M. I. Schwartzbach. A type system for dynamic Web documents. In *Symposium on Principles of Programming Languages*, pages 290–301, 2000.

21. Thiemann, P. WASH/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Practical Applications of Declarative Languages*, pages 192–208, 2002.