

Web Interactions

A Dissertation Presented

by

Paul Thorsen Graunke

to

The College of Computer Science

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in the field of

Computer Science

Northeastern University

Boston, Massachusetts

June 2003 A.D.

© June 2003 A.D.

Paul Thorsen Graunke

ALL RIGHTS RESERVED

Web Interactions

by

Paul Thorsen Graunke

ABSTRACT OF DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate School of Computer Science
Northeastern University, June 2003 A.D.

ABSTRACT

This dissertation focuses on programming language support for interactive dialogues that exhibit the same flexibility as Web dialogues. A foundational model of Web interactions provides a framework for formally describing two classes of errors. The model suggests techniques for detecting both classes of errors. An incrementally checked record type system effectively eliminates one class of errors. A dynamic safety check catches the other class of errors relative to programmers' simple annotations.

In practice, programming with continuations greatly facilitates the implementation of Web dialogues. This dissertation explores two techniques for extending programming languages with Web interaction primitives based on continuations. The first technique relies on a custom Web server that acts as a high-level operating system for Scheme servlets. Managing both server and servlets in one process results in good performance and easy communication between servlets. The second technique relies on whole-program transformations often found in compilers for functional programming languages. This results in standard CGI programs that run on any Web server, but it does not assist communication between transformed Web programs or facilitate placing limitations on backtracking.

Web and GUI programs represent two extremely common and popular modes of human-computer interaction. Many GUI programs share the Web's notion of *browsing* through data- and decision-trees. This dissertation also compares the user's browsing power in the two cases and illustrates that many GUI programs fall short of the Web's power to clone windows and bookmark applications. It identifies a key implementation problem that GUI programs must overcome to provide this power. It then describes a theoretically well-founded transformation that endows GUI programs with these capabilities. Concrete examples demonstrate the transformation in action.

Acknowledgments

I owe many thanks to many people for their support during my journey as a PhD student. I could not possibly thank everyone who deserves my gratitude, but much of life is not about deserving anyway. Since you, gentle reader, have opened this document, there is a fair chance that you too are a graduate student or will become one someday. Remember, the people in your life during graduate school are at least as important as your studies.

Matthias Felleisen sacrificed much for us, his students. Working 100% of his time on many projects simultaneously, his passion for excellence and his drive to overcome obstacles of any magnitude has left a lasting impression. I'm grateful not only for his quest to discover better software construction methodology but also for his eagerness to spread the wealth to all who will listen.

Shriram Krishnamurthi first excited me with the beauty of Scheme in my freshman computational science lab at Rice. Since then he has exerted much patient-endurance in training me, not only in the ways of functional programming, but also in presenting ideas. Few can match his flair.

Robby Findler's companionship, attention to detail, and willingness to be blunt when needed made the low points of graduate school much more bearable.

Matthew Flatt stood out in many ways. His ability to produce crisp designs and volumes of dependable code while working 12 hour days amazingly left time for golf, television, and camping. How he manages, I may never know.

My parents always invested in my education from an early age. Although opening a college

fund when I was six months old and later driving me to various science events concretely provided opportunities for me to learn, the underlying message of education's importance and their care for me were at least as important.

I'm grateful for the Church, whose kindness, encouragement, and prayers have brightened my days. The joy and passion of many in the Houston Vineyard spurred my desire for more of Christ's Spirit dwelling in me.

Reece Price's wise counsel, steady friendship, and joyful accounts of God's work in his life always lifted me up. I deeply appreciate him and his wife Pat for their prayers on behalf of my mother during her chemotherapy.

Craig Johnson, fellow believer, graduate student, and former roommate brightened many an evening and helped keep life in perspective. Understanding graduate life first-hand while remaining detached from technical matters in my field made Craig ideally suited to lend an ear.

Jesus Christ, the only one not deserving death, bore the just penalty for my sin through His crucifixion, death, and burial. The blameless and everlasting life I have before God by Christ's resurrection and return to heaven made this dissertation possible.

S.D.G.

Contents

Abstract	iii
Acknowledgments	iv
1 Introduction	1
1.1 A Sample Web Interaction Error	1
1.2 Web Programming Background	5
1.3 Contributions	6
1.4 Prior Publication of Contributions	8
2 Related Work	9
2.1 Web Programming	9
2.1.1 Supporting Web Navigation	11
2.2 Types and Modeling	15
2.3 Performance	15
2.4 Resource Management	16
2.5 GUI Interactions	18
2.6 Inverting Event Driven Programs	19
3 Serving the Web	23
3.1 Web Servers and High-Level Operating Systems	23
3.2 MrEd: a High-Level Operating System	24

3.3	Serving Static Content	26
3.3.1	Implementation of the Web Server's Core	26
3.3.2	Performance	27
3.4	Dynamic Content Generation	28
3.4.1	Simple Scheme Servlets	29
3.4.2	Content Generators are First-Class Values	31
3.4.3	Exchanging Values between Content Generators	31
3.4.4	Interactive Servlets	32
3.4.5	Implementing Interactive Servlets	35
3.4.6	Performance	38
3.4.7	Modularity of the Server	38
4	Compiling to the Web	41
4.1	Designing Web Programs	41
4.2	Interactive CGI Programs	43
4.2.1	Conventional CGI Programs	43
4.2.2	Direct-Style CGI Programs	46
4.3	Generating CGI Programs	47
4.3.1	Functional CGI Programs	48
4.3.2	Compiling Stateful CGI Programs	53
4.4	Developing CGI Scripts	55
4.5	Implementation Status	57
5	Modeling the Web	61
5.1	Introduction	61
5.2	Modeling the Web	61
5.2.1	Server and Client	62

5.2.2	Functional Web Programming	64
5.2.3	Stateful Web Programming	64
5.3	Problems with the Web	65
5.3.1	The Communication Problem	66
5.3.2	The Observer Problem	67
5.4	Type Checking Communication	68
5.5	Notifying Outdated Observers	76
6	Beyond the Web	81
6.1	Introduction	81
6.2	Usage Patterns in Interactive Programs	83
6.2.1	Interaction Diagrams	84
6.2.2	Desiderata	87
6.2.3	Implementation Challenges	89
6.2.4	Stack Patterns in Flexible GUIs	90
6.3	Implementing Flexible GUIs: Control	97
6.3.1	Transforming GUI Programs for Flexibility	97
6.3.2	Two Dimensions of Complexity	103
6.4	Implementing Flexible GUIs: Data	104
7	Conclusions	106
7.1	Web Dialogues via a Server	106
7.2	Web Dialogues via Compilation	107
7.3	Model of Web Dialogues	108
7.4	GUI Dialogues	109
A	Expressiveness of send/suspend	121

List of Figures

1.1	Orbitz Interactions	2
1.2	iCard	4
3.1	MrEd's TCP, thread and custodian primitives	25
3.2	Performance for static content server	27
3.3	X-expression Responses	30
3.4	Additional content generator primitives	33
3.5	An interactive servlet	34
3.6	Comparison of Interaction Primitives	35
3.7	Server Resources	37
3.8	Performance for dynamic content generation	39
4.1	An Interactive Addition Program	44
4.2	A CGI Version of Figure 4.1	44
4.3	The Compiled Version of Figure 4.1	48
4.4	The CGI Version of Figure 4.3 (Compare with Figure 4.2)	50
4.5	Lambda Lifted	51
4.6	A Stateful Interactive Program	58
4.7	Its CGI Version	58
4.8	CGI Error Reporting	59
4.9	CGI Stepping	60

5.1	The Web	62
5.2	The Web Picture	63
5.3	Transitions	63
5.4	Web Programming Language	65
5.5	Language Extensions for Storage	66
5.6	Collaborating Programs	67
5.7	Stateful Web Programs	68
5.8	Internal Types for WrForm	70
5.9	Constraint Checking	72
5.10	Outdated State in WASH/CGI	77
5.11	Relevance for Storage	78
5.12	Stateful Web Programs	79
5.13	Reductions	79
5.14	Avoiding Outdated Flights	80
6.1	Inflexible Interface Version	91
6.2	Version with Backtracking Support	94
6.3	Transformed Version	99
6.4	Multiple Submissions in Action	102
6.5	Bookmarking in Action	103
A.1	Form Functions For Cawl-cc	125

Chapter 1

Introduction

Over the past decade, the Web has become an interactive medium. Far more than half of all Web transactions are interactive [20]. 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.

1.1 A Sample Web Interaction Error

The complexity of programming Web applications all too often results in noticeable software defects. A example from a commercial Web site illustrates a common error users encounter on the Web. Figure 1.1 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

¹The screenshots were produced on June 28, 2002. The problems persisted when last checked, on October 24, 2002.

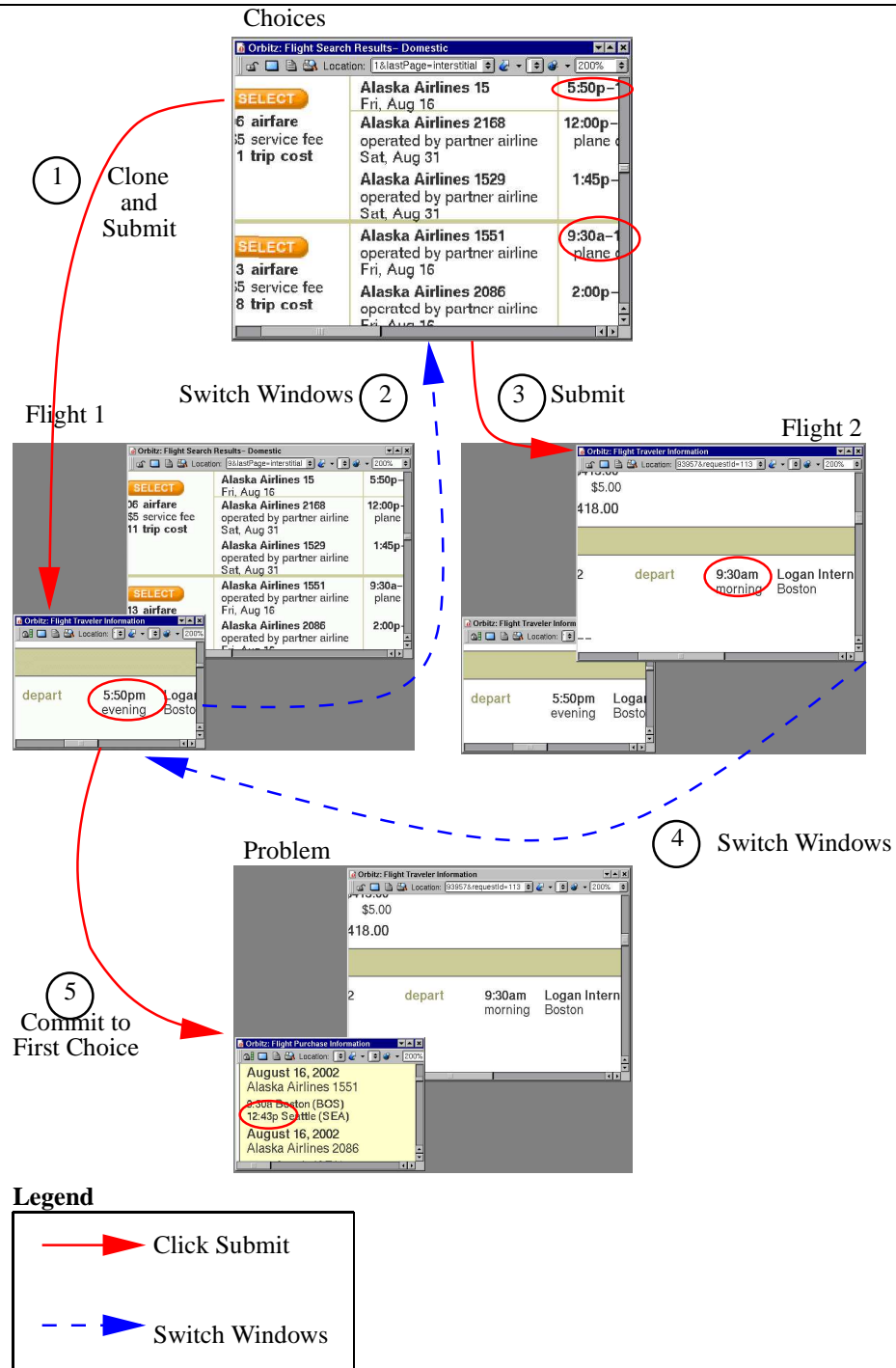


Figure 1.1: Orbitz Interactions

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. Alarming, 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 the point. 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, my colleagues and I have stumbled across this and related problems while using several vendor's sites, including Apple, Continental Airlines, Hertz car rentals, Microsoft and Register.com.

Figure 1.2 illustrates another commercial instance of the problem. Apple's iCard service allows consumers to send electronic greeting cards via email. The service suffers from several common problems, which the following example scenario illustrates.

To use the site, the consumer first chooses a picture for the card. Next, the consumer attaches a message to the card. The site can send the card to several recipients. Each recipient is added to the list separately. Here the consumer enters just one email address and sends the card.

Seeing the confirmation page, the consumer switches to another other task—reading email. Unhappily, a mail message states that the card could not be delivered due to an invalid email address. To correct the problem, the consumer clicks the browser's back button, fixes the recipient's email address, and resubmits the form. Another confirmation page appears.

Checking for new email, the consumer again sees a message indicating the system could not send the card to the invalid email address. In fact, the card was sent to *both* the mistaken and corrected email addresses, revealing the embarrassing mistake to the card's recipient. Clicking

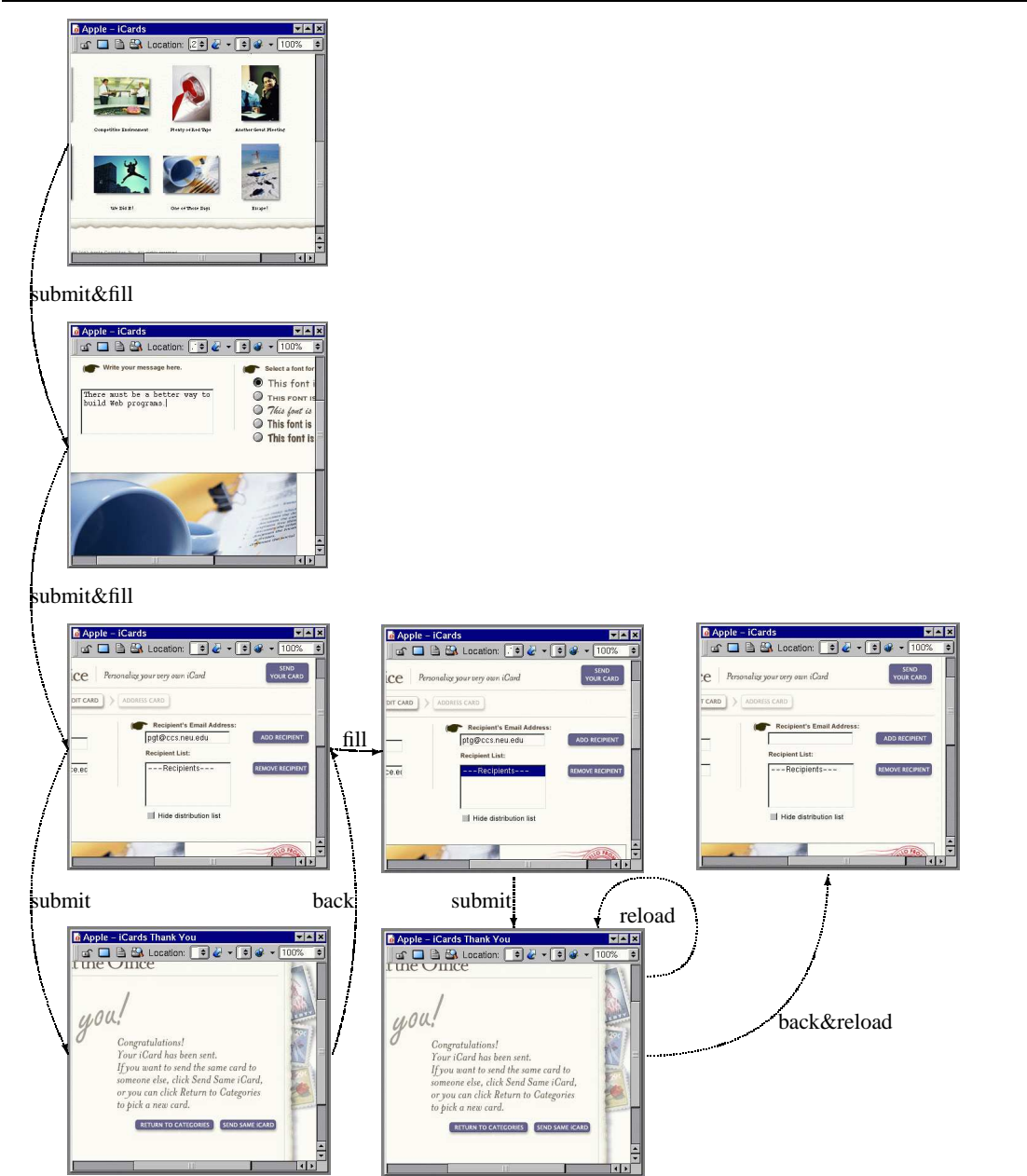


Figure 1.2: iCard

reload on the confirmation page only makes matters worse. The program sends the same card to both email addresses again for further embarrassment.

After some thought, the consumer realizes the page with the corrected email address does not accurately reflect the state on the server. In particular, the list of additional recipients appears empty while the list on the server still contains the incorrect address. Clicking the back button and reloading the recipient page, however, does not reveal the updated recipient list. Instead the program clears the recipient list on the page and on the server.

Clearly, an error that occurs repeatedly across organizations suggests not a one-time programming fault but rather a systemic problem.

1.2 Web Programming Background

To understand these problems, 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 (Common Gateway Interface (CGI) [82], Java servlets [29], or Microsoft's ASP.NET [77]). 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 a Hyper-Text Transfer Protocol [42] (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 with applications 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.

The shortcomings of HTTP lead to other challenges for Web programmers. Consumers start and quit non-Web applications to control resource consumption. Web applications cannot rely on this since quitting a Web browser, closing its windows, or deleting bookmarks do not inform the server of the user's actions. The limited communication between Web servers and browsers also prevents Web programs from updating out-dated information on the client.

1.3 Contributions

This dissertation addresses the difficult nature of Web programming, such as the above examples, in several ways. After chapter 2 discusses related work, chapter 3 presents some programming language extensions that drastically simplify the construction of Web programs and help avoid programming errors in practice by preserving a Web program's environment between interactions. A custom Web server implements these extensions by dynamically loading Web programs and managing them using MrEd's [48] operating-system (OS) constructs. Chapter 4 explores an alternative implementation based on compilation techniques, which move the preserved environment to the client.

Chapter 5 makes three contributions to the problem of designing reliable interactive Web programs. First, a simple but formal model of Web interactions provides a framework for explaining the above problems concisely and can also study schematic solutions. Second, a type system captures the checks this model suggests. Third, because not all the checks can be

performed statically, a run-time check maintains store consistency. The run-time check detects the situations in section 1.1, which is useful for maintaining non-backtracking state.

Chapter 6 compares the user interface provided by Web applications with those of installation or setup programs. Both kinds of applications dialogue with the consumer through a series of graphical forms. Web applications, however, sport more flexible interfaces. While this improved flexibility challenges the implementor, it eases the consumer's burden of using the program.

Utilities such as GUI-building wizards only minimally assist interactive software developers. Most of these programs largely deal with the tedious task of laying out and instantiating visual elements. They still leave the programmer to deal with the difficult problems of designing and composing the actual behaviors that lie beneath the "callbacks". Unfortunately, most of the difficult tasks lie in the *interaction* between the visual elements and the underlying behavior. As a result, the wizards do not address many of the truly hard problems that arise in writing GUI programs.

Many installers' GUIs provide half-baked mechanisms for adjusting earlier decisions. Implemented in an ad hoc fashion, backtracking within installers often breaks or imposes severe limitations. When faced with these difficulties, consumers must resort to more drastic measures such as restarting the program or entire computer possibly after deleting partially installed files.

When installing or configuring software, consumers often face uncertainty. Choices made during the initial part of the dialogue can lead to unknown ramifications much later. In these cases the consumer may wish to explore several options side-by-side. After exploring the implications of the decision, the consumer should be able to abandon all but one of the branches.

Another common task network administrators face when installing software involves performing the same or similar installations on many machines. In these situations some installation programs offer the ability to read installation settings from a configuration file. This enables administrators to re-use the same configuration on multiple machines. When present, the implementation of extracting a configuration from a file is completely separate from that of extracting

a configuration from a dialogue with the user. Not only does the implementor need to create, document, and maintain two separate user interfaces, the user must learn two completely separate mechanisms for specifying a configuration.

All of the above problems that installer implementors face benefit from lessons learned in constructing Web applications. Chapter 6 presents a systematic transformation for adding Web-like features to the kinds of dialogues found in installers. These extensions allow consumers to change previous choices in a dialogue, explore several possibilities in parallel, and bookmark the current progress in the dialogue.

1.4 Prior Publication of Contributions

Portions of this dissertation appeared in prior publications. Chapter 3 presents a slightly updated version of “Programming the Web with High-Level Programming Languages” [61]. Chapter 4 is based on “Automatically Restructuring Programs for the Web” [59]. The results from chapters 3 and 4 also appeared in the author’s master’s thesis [58]. Chapter 5 presents the model found in “Modeling Web Interactions” [60]. Chapter 6 extends the work to non-Web dialogues, as in “Advanced Control Flows for Flexible Graphical User Interfaces or, Growing GUIs on Trees or, Bookmarking GUIs” [62].

Chapter 2

Related Work

As no work stands in isolation, my dissertation both builds on and contrasts with similar work addressing related problems. Many other systems address some of the issues raised by Web programming. Linguistic support for constructing special purpose operating systems appears in other contexts as well. Several languages rely on continuations [45, 88, 103], to simplify the task of programming network applications. Ever since the formative years of the λ -calculus, types have played an important roll in preventing errors in programs.

2.1 Web Programming

The enormous volume of systems designed for Web programming allows this section to only touch on some of the more popular and interesting ones. Widely-used commercial systems are either script-centric or page-centric. Script-centric technologies, such as CGI or Java servlets, always invoke a program or method from the beginning—without any (helpful) control context. This prevents the individual scripts comprising a dialog from communicating simply and reliably through common lexical scope and shared storage. Page-centric systems such as Java Server Pages (JSP) [95], Active Server Pages (ASP.NET), and PHP [13] face all the problems that script-centric approaches do and also commit to much of the page's appearance before processing any form inputs. This leads to contorted programming patterns involving scripts redirecting requests in order to produce pages with a different look in the event of an error.

In light of the difficulties communicating values between Web scripts, some Web technologies provide server-side storage that maintains values across interactions. Aside from the object-oriented interface and libraries for constructing HTTP response headers, servlets provide almost the same programming model as standard CGI. Each incoming request invokes a `doGet` or `doPost` method in the servlet from the beginning, leaving the task of restoring the appropriate control context to the programmer. It may appear that servlets can avoid moving the storage into cookies by storing values in the servlet object's fields. The Java servlet specification explicitly forbids this, however, since the server (servlet container) can unload the servlet at anytime and reload a fresh copy to handle the next request. The server also has the option of migrating the servlet to another virtual machine, so data may not reside in static fields between interactions either. Servlets assist programmers with this problem by providing `HttpSession` objects that contain methods for maintaining a dictionary from strings to Objects on the server and storing a reference to the dictionary in a Uniform Resource Locator (URL), cookie, or Secure Sockets Layer session. Active server pages also provide objects for storing session, application, and global data, which provide different levels of sharing. Although session objects eliminate the need to marshal data into and out of hidden fields or cookies, the information they store is not sensitive to control-flow operations induced by browser's navigation capabilities. They suffer from timeouts, as do most forms of server-side state. These forms of server side state can also become unsynchronized with respect to information visible on old Web pages.

At first glance, a reader might suspect that the FastCGI protocol solves the problems of engineering CGI programs by explicitly waiting for a request in the middle of the program. The FastCGI protocol starts a separate process on the server for each Web program. The server forwards successive requests to the FastCGI program, which sends the responses back to the server. Since these programs wait for a request, it appears at first that the programmer could do more than the typical looping over requests at the start of the program. One could attempt to construct an interactive program by waiting for the next request at different points in the

computation. This approach, however, only allows the user to proceed forward through each interaction. Cloning windows or using the back button will send the form data to the wrong point, causing the FastCGI program to either not find fields expected from the correct form or, even worse, to misinterpret fields that accidentally coincide.

The “Mother of All Web Languages” (MAWL) system [12] uses this idea of a thread waiting for requests at different points in the code to transparently preserve program state across interactions. Since previous pages representing old program state are no longer accessible, users must restart transactions to correct mistakes. Their experience indicates that users complained about this inability to use the back button or the browser’s page history.

The `<bigwig>` project [18, 99], a descendent of MAWL, partially addresses the issue of connecting the control flows of individual Web scripts. In `<bigwig>`, the expression

$$\text{show } d \text{ receive } [v_1 = f_1, \dots, v_n = f_n]$$

sends a Web form to the client and then waits for the consumer to fill out and submit the form. The variables $v_1 \cdots v_n$ bind to the values of the submitted form’s fields. If combined with a decent core programming language, this construct in principle reduces the programmer’s problem of constructing Web programs to that of constructing sequential command-line programs. Regrettably, the resulting programs behave no more flexibly than the command-line programs they mimic. That is, they fail to support the browser’s advanced navigation features since 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.

2.1.1 Supporting Web Navigation

The starting point for constructing direct style programs that respect backtracking is Jackson’s notion of program inversion [66]. When a program produces tree-shaped data and another program consumes the data, the programmer can intertwine the two computations by changing

the two functions into coroutines. Although developed to conserve memory or reduce the number of intermediate tapes used by Cobol programs, the switching between coroutines matches the switching between the program waiting for the user to fill out a form and the user waiting for the program to compute the next page.

Several Web systems do properly support the browser's backtracking, cloning, and book-marking features. By preserving former coroutine resumption points, a Web program can return to these points in the program's execution multiple times. John Hughes [64], Christian Queinnec [91], and Paul Graham [56] independently had the 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. With explicit access to continuations, instead of terminating, a program can capture its continuation and suspend at an interaction point. Every time a consumer submits responses, the computation resumes the proper continuation. Not only does internalizing the communication among scripts simplify the program, it also subjects the communication to the safety mechanisms of the language.

Continuation passing style (CPS) is commonly used in compilers for advanced functional languages [9, 101]. While CPS is normally used by compilers to name intermediate terms (as observed by Sabry [96, 97]), it has the effect of making the continuations in the program explicit. Prior work on CPS can also guide the implementation of the transformer on control features such as exceptions [16]. A similar extension of the transformer handles multiple submission options in the form of *double-* and, in general, *multi-barrel* continuations [108].

Programmers building imperative-style programs in purely functional languages use a technique based on the mathematical theory of monads. Hughes [64], in developing his theory of arrows, a generalization of monads, describes how to implement interactive CGI programs using arrows. His key insight is to provide a mechanism that at each interaction point turns the current continuation into a datum for the Web page. This requires an operation on continuations

not supported by most languages with continuations. His work properly handles backtracking, but it does not provide mutable storage which remembers information despite backtracking.

Thiemann [109] used Hughes's ideas as a starting point and provides a monad-based library for constructing Web dialogs. In principle, his solution corresponds to the compilation-based implementation in chapter 4; his monads take care of the "compilation" of Web scripts into a suitable continuation form. A later version of Thiemann's Web Authoring System Haskell/CGI (WASH/CGI) library [83] uses 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 by adding type-indexed state and by remembering Input/Output (I/O) operations. Specifically, for each interaction, his CGI scripts are re-executed from the beginning to the current point of interaction. Even though this avoids the re-execution of effects, it is indicative of the problems with Thiemann's approach. Like this dissertation's compilation solution, Thiemann's approach won't easily apply to other languages.

Similarly, Queinnec [91] advocates using *call/cc* to implement interactions between Web servers and consumers. His method requires the modification of a server that can store continuations. It does not, however, adequately address resource management issues.

This dissertation follows up on Queinnec's work and explored its implications in two ways. First, its experimental Web server enables Web programs to interact directly with consumers [61]. Programming in this world eliminates many of the Web design problems in a natural manner. Second, since this solution doesn't apply to languages without such mechanisms, a second approach explores the automatic generation of robust Web programs via functional compilation techniques [59]. While this idea works in principle, a full-fledged implementation requires a re-engineered library system and runtime environment for the targeted language (say Perl [111] or Python [110]).

Graham [56, 57] claims that the success of his Viaweb company, now Yahoo! Shopping, is

due in part to the methodical use of continuation-passing-style to construct Web applications. If this technique proves helpful when done semi-manually using Scheme [69] macros [27], then using an automated translation must be even better. He does not explain how his company dealt with mutable storage.

The Second Interpreter of Scheme Code (SISC) [78] could support continuation-based Web programming. The implementation compiles Scheme to Java, including full support for continuations. Since the continuations are serializable, “Java” servlets written with SISC could store their control state on Web pages, similar to the technique described in chapter 4.

The Seaside [21] framework for constructing Web programs in Smalltalk [54] also supports a session programming model using continuations. As with everything in Smalltalk, activation records are objects that inherit from the class `Object`. Under some of the more correct implementations, this allows programs to copy their stack using the activation record’s `clone` method inherited from class `Object`. These copied activation records function similarly to continuations, and can support browser-directed backtracking. The common mutation-filled, object-oriented paradigm used in Smalltalk programs, leads to some questions as to whether or not instance fields should be restored during backtracking. Seaside 2.2 requires programmers to wrap backtracked fields in `StateHolder` objects.¹ The library enables programs to delimit the set of pages that may be returned to via the `interact` method. It is not clear that the server can shutdown Web programs or protect the entire server from damage.

All of these systems permit Web programs to result in confusing mismatches between information on a Web page and the information on the server. Neither do they provide a model of Web interactions.

¹Perhaps they should be named “`EnvironmentHolder`” objects instead.

2.2 Types and Modeling

Programmers have used type systems to eliminate errors from programs since the early days of computation science [86, p.2]. Using a typed language does not in itself, however, prevent errors. Even “strongly-typed” languages such as Meta Language (ML) [79] must allow for runtime checks to distinguish between variants. Ocaml’s CGI library [43] represents Web requests as association lists of strings, mapping field names to values. Thus, the type checker cannot distinguish between different forms at all.

Other formal systems such as those for mobile agents tend to be overly flexible for the purpose of modeling Web interactions. Web programs typically do not migrate between machines or nest inside each other as ambients do [23].

2.3 Performance

The performance problems of the CGI interface has led others to develop higher-speed alternatives. In fact, one of the driving motivations behind the Microsoft .NET initiative [77] appears to be the need to improve Web server performance, partially by eliminating process boundaries.² Apache provides a module interface [107] that allows programmers to link code into the server that, among other things, generates content dynamically for given URLs. However, circumventing the underlying operating system’s protection mechanisms without providing alternative protection within the server opens the process for catastrophic failures.

FastCGI [84] provides a safer alternative by placing each content generator in its own process. Unlike traditional CGI, FastCGI processes handle multiple requests, avoiding the process creation overhead. The use of a separate process, however, generates bi-directional inter-process communication costs, and introduces coherence problems in the presence of state. Communicating higher-order data through character-based communication ports is also difficult.

IO-Lite [85] demonstrates the performance advantages to programs that are modified to

²Jim Miller, personal communication with Shriram Krishnamurthi.

share immutable buffers instead of copying mutable buffers across protection domains. Since the underlying memory model of MrEd provides safety and immutable string buffers, the server of chapter 3 automatically provides this memory model without the need to alter programming styles or Application Programming Interfaces (APIs).

Java servlets address performance issues by avoiding both the process creation overhead (as does FastCGI) and the interprocess communication overhead. Loading all the servlets into the same process as the server enables the efficient exchange of Web requests and responses. This solution, however, raises resource management issues.

2.4 Resource Management

In addition to supporting the flexible control flows demanded by modern Web browsers, Web development environments must also manage the resources of Web programs. The Scheme Web server in chapter 3 uses the resource management primitives provided by MrEd as does DrScheme-esq [48]. Other work on resource containers [15] provides a similar, more complex, mechanism for separating the ownership of resources from traditional process boundaries.

Like the programs that extend the Scheme server, Java servlets are content generating code that runs in the same runtime system as the server. Passing information from one request to the processing of the next request cannot rely on storing information in instance variables since the servlet may be unloaded and re-loaded by the server. Passing values through static variables does not solve the problem either, since in some cases the server may instantiate multiple instances of the servlet on the same Java Virtual Machine (JVM) [74] or on different ones. Instead they provide session objects that assist the programmer with the task of manually marshaling state into and out of re-written URLs, cookies, or Secure Socket Layer connections.

The Java servlet interface allows implementations to distribute requests to multiple JVMs for automatic load balancing purposes. Lacking the subprocess management facilities of MrEd's custodians, they rely on an explicit delete method in the servlet to shut the subprocess down

cooperatively. While this provides more flexibility by allowing arbitrary clean-up code, the servlet isn't guaranteed to comply.

The Java community realizes the shortcomings of the deprecated `Thread.stop()` method. Accordingly, J-Server [100] extends an implementation of Java to prevent the servlets from shutting down the entire server while allowing the server to reliably shutdown the servlets. J-Server runs atop Java extended with operating systems features. In order to communicate with each other, their servlets must rely on an Remote Method Invocation (RMI) protocol, which is less efficient, leads to more coherence concerns, and is less convenient to program than sharing values through lexical scope. Their work addresses issues of resource accounting and quality of service, which is outside the scope of this dissertation. Their version of Java lacks a powerful module system and first-class continuations.

The problem of managing resources in long-running applications has been identified before in the Apache module system [107], in work on resource containers [15], and elsewhere. The Apache system provides a pool-based mechanism for freeing both memory and file descriptors *en masse*. Resource containers provide an API for separating the ownership of resources from traditional process boundaries. The custodians in MrEd provide similar functionality with a simpler API than those mentioned.

Development Environment

The Java Platform Debugger Architecture [105] enables Java development environments [17, 65, 106] to attach remotely to the JVM that the Web server uses to run servlets. The professional and enterprise editions of the JBuilder development environment [17] integrates with a Web server to provide better feedback while debugging Java servlets. Although this reuses existing development environments to debug Web applications by setting break points and displaying the source of exceptions, it does not assist the programmer with the convoluted structure of interactive servlets.

2.5 GUI Interactions

Some related work develops novel methods of constructing GUIs without proposing new forms of user interaction. Other works provide alternate implementations of continuations on Java virtual machines without discussing GUIs at all. Earlier systems provided a mechanism for saving and resuming program state without separating the two kinds of information flow.

Elliot and Hudak's functional reactive animation [39] supports an unusual style of creating event-driven animations without explicitly dealing with timing details. In their system programmers define models of animation in terms of events that include the elapsing of specific amounts of time. The work focuses on a smaller-grained problem of handling graphics within windows rather than transitions between windows.

GUIs written using Fudgets [24] also exhibit an unusual program structure. Programmers compose functions representing graphical components to create larger functions that thread streams of events from one component to the next. Although this may facilitate combining components, programmers must still connect events from one component to the next manually. The work does not attempt to provide more flexible interfaces to users.

Fuchs's Dreme [51] provides a mobile code system for Scheme. It also addresses mobile user interfaces, and claims that callbacks in event-driven programs are a "twisted" form of continuations. The work does not address bookmarking GUIs, nor does it investigate new kinds of user interactions.

Most exception-based implementations of continuations on Java Virtual Machines fail to provide the flexibility of multiple resumptions this work requires. However, Fünfroeken [52] and Sakamoto, et al. [98] both provide an adequate mechanism for capturing and resuming control state on Java Virtual Machines using a global program transformation in conjunction with exceptions. Programmers could use their techniques to implement GUIs supporting parallel exploration, but they do not suggest doing so, nor does it clearly extend to bookmarking.

The flexibility my dissertation lends to GUIs originates from the navigational capabilities of Web browsers. The same continuation-based techniques for designing and implementing Web programs apply to GUI dialogues that support user-directed backtracking.

The bookmarking portion of this work relies on Java serialization for object persistence. An object-oriented database system such as JavaSPIN [114] could replace serialization to better manage saved bookmarks. Packaging all of the remaining computation into a button callback greatly facilitates saving and restoring the user's interactions. Persistent storage alone, however, does not enable parallel exploration or the ability to complete an operation multiple times.

Various programming environments for Common Lisp [102] and for Smalltalk [54] allow the user to save the entire application state and resume the program at a later time. This facility does not support parallel exploration easily since saving, quitting, and restarting the entire application for each switch between exploration branches is too cumbersome. This approach also copies any state shared between interactions.

Recent work by Zandy and Miller [118] extends their application migration framework with the ability to migrate individual application's X windows sessions. The system uses an extended X server to marshal the current snapshot of an application's windowing state. The GUI migration program can then reconstruct the windowing state on another extended X server, while hiding any internal differences from the application. The paper points out the advantages of migrating individual applications instead of entire desktops. Chapter 6 of this dissertation allows even finer control by migrating individual steps in a dialogue. Zandy's work does not address parallel exploration or backtracking. On the other hand it does solve many low-level problems such as translating window ids.

2.6 Inverting Event Driven Programs

The Teapot language [25] provides a weak form of continuations to ease the task of constructing cache coherence protocols. The protocol implementations maintain a state (chosen from a finite

set) for each cache block. Transitions between states occur based on the current state and on messages received from the network. To avoid deadlock, the code that implements these transitions must run to completion and terminate—much like CGI programs. Without the benefits of Teapot, programmers need to introduce extra states that record the protocol’s progress in the middle of multi-message transitions. Teapot provides *Suspend* and *Resume* operations to temporarily enter a transient state and later resume execution in the middle of the suspended message handler.

The Teapot system refers to these captured control states as continuations, although they differs from Scheme’s continuations [69]. Calls to *Suspend* and *Resume* must pair up like they do for normal function calls and returns. Also, programs may not call *Suspend* inside procedure calls; all *Suspend* calls must occur directly within top-level message handlers.

Although Teapot programs face some unexpected execution orderings as do Web programs, the ordering problems derive from a different source. Network protocols for cache coherence face the annoyance of receiving out-of-order network packets. Unusual orders that would not otherwise occur can be viewed as mistakes. Teapot provides a mechanism for correcting these mistakes by queuing the out-of-order messages and handling them later. In contrast, unexpected execution orders in Web programs arise from the consumer’s willful use of the browser’s navigation features. Refusing to respond to a Web request until the program receives a request it expects would infuriate consumers.

Despite Teapot’s continuations not enabling flexible interactions or programs which suspend during procedure calls, these limitations help Teapot achieve its goal of producing robust finite state machines. The lack of dynamic memory allocation enables Teapot to compile its programs to the Mur Φ verification system. This dissertation’s focus on general purpose programs with interactive procedure calls and dynamic memory allocation make software verification more difficult and beyond the scope of this work.

Some systems programmers create event driven programs, they essentially re-invent their own

thread primitives and scheduler. Instead of using blocking I/O operations, the single threaded program invokes asynchronous I/O operations and enqueues some representation of the control state (i.e. an ad hoc continuation). The program then calls the `select` system call to see which suspended “threads” are runnable. It reschedules the “thread” by dequeuing and invoking the “continuation”.

Others researchers [6, 115] are also rediscovering how continuations can implement cooperative threads [112]. Although threads do block for I/O operations, they are not suitable for implementing Web applications since a suspended thread cannot resume multiple times from the same I/O operation. C’s *setjmp* and *longjmp* operations are likewise unsuitable.

Brian Demsky’s masters thesis [34] compares three styles that servers use to handle concurrent network connections. The simplest way to handle multiple connections is to start a new thread for each connection. This is the technique used in chapter 3. Some servers maintain a pool of threads to avoid the overhead of creating many new threads. Demsky’s thesis shows that thread pools result in substantial savings, although more recent work [36] suggests this is less of an issue with an improved thread implementation. The last implementation considered is an event driven style, popularly touted for its performance.

The primary contribution of Demsky’s work is to analyze why event driven programs perform well and compile the direct-style thread-per-connection programs into equivalent event driven programs. The main performance benefits come from avoiding thread creation overhead and avoiding synchronization costs. By only switching threads during I/O operations or explicit yields, threads can execute (I/O free) critical sections without locking.

Although Demsky’s work compares event driven programming to direct-style programming and provides a automated translation from direct, thread-per-connection style to event driven style, its goals differ from those of this dissertation. Without Demsky’s transformation, programmers could still construct servers using the thread-per-connection style; the servers just would not execute as fast. The server invokes each continuation only once; unusual control flows

do not emerge. Specifically, the transformed programs do not support a browser's navigation features. Resource management and a model of Web interactions are outside the scope of his work.

Chapter 3

Serving the Web

3.1 Web Servers and High-Level Operating Systems

A Web server provides services similar to those of operating systems. Like an operating system, a server runs programs (e.g., Web scripts). Like an operating system, a server protects these programs from each other. And, like an operating system, a server manages resources (e.g., network connections) for the programs it runs.

Some existing Web servers rely on the underlying operating system to implement these services. Others fail to provide services due to shortcomings of the implementation languages. This chapter shows that implementing a Web server in a suitably extended functional programming language is straightforward and satisfies three major properties. First, the server delivers *static* content at a performance level comparable to a conventional server. Second, the Web server delivers *dynamic* content at five times the rate of a conventional server. Considering the explosive growth of dynamically created Web pages [20], this performance improvement is important. Finally, the server provides programming mechanisms for the dynamic generation of Web content that are difficult to support in a conventional server architecture.

The basis of this experiment is MrEd [48], an extension of Scheme. The implementation of the server heavily exploits four extensions: first-class *units* of code, which help structure the server and represent dynamic server extensions; preemptive *threads*; which are needed to execute server programs; *custodians*, which manage the resource consumption of server programs;

and *parameters*, which control stateful attributes of threads. The server programs also rely on Scheme’s capabilities for manipulating continuations as first-class values. This chapter shows which role each construct plays in the construction of the server.

The following section is a brief introduction to MrEd. Section 3.3 explains the core of the server implementation. Section 3.4 shows how the extended server dynamically generates Web content via Scheme servlets and illustrates how programming in the extended Scheme language facilitates the implementation of these scripts. Sections 3.3 and 3.4 also present performance results.

3.2 MrEd: a High-Level Operating System

MrEd is a safe implementation of Scheme; it is one of the fastest existing Scheme interpreters.¹ Following the tradition of functional languages, a Scheme program specifies a computation in terms of values and legitimate primitive operations on values (creation, selection, mutation, predicative tests). The implementation of the server exploits traditional functional language features, such as closures and standard data structures, and also Scheme’s ability to capture and restore continuations, possibly multiple times.

MrEd extends Scheme with structures, exceptions, and units. The unit system [47] permits programmers to specify *atomic units* and *compound units*. An atomic unit is a collection of definitions. Each unit has an import and an export signature. The import signature specifies what names the unit expects as imports; the export signature specifies which of the locally defined names will become visible to the rest of the world. Units are first-class values. There are two operations on unit values: invocation and linking. A unit is invoked via the *invoke-unit/sig* special form, which must supply the relevant imports from the lexical scope. MrEd permits units to be loaded and invoked at run-time. A unit is linked—or *compounded*—via the *compound-unit/sig* mechanism. Programmers compound units by specifying a (possibly cyclic)

¹Native code compilers tend to perform better, though.

```

tcp-listen : Nat [Nat]  $\longrightarrow$  Tcp-listener
;; reserves a port to accept connections, optionally specifying the
;; maximum number of clients that may wait for a connection

tcp-accept : Tcp-listener  $\longrightarrow$  * Input-port Output-port
;; creates I/O ports for a connection request via the listener

```

```

thread : ( $\longrightarrow$  Void)  $\longrightarrow$  Thread
;; spawns a thunk as a thread

make-semaphore : Nat  $\longrightarrow$  Semaphore
;; creates a semaphore with specified number of tokens

semaphore-post : Semaphore  $\longrightarrow$  Void
;; posts a semaphore and releases waiting threads

semaphore-wait : Semaphore  $\longrightarrow$  Void
;; waits (and possibly suspends) for a semaphore

```

```

make-custodian :  $\longrightarrow$  Custodian
;; creates a custodian

custodian-shutdown-all : Custodian  $\longrightarrow$  Void
;; shuts down all threads in custodian and reclaims all resources

```

Figure 3.1: MrEd’s TCP, thread and custodian primitives

graph of connections among units, including references to the import signature; the result is a unit.

The extended language also supports the creation of threads and thread synchronization. Figure 3.1 specifies the relevant primitives. Threads are created from 0-ary procedures (thunks); they synchronize via counting semaphores. For communication between parent and child threads, however, synchronization via semaphores is too complex. For this purpose, MrEd provides (thread) *parameters*. The form

(**parameterize** ($[parameter_1\ value_1]\dots$) *body* \dots)

sets $parameter_1$ to $value_1$ for the dynamic extent of the computation $body\ \dots$; when this computation ends, the parameter is reset to its original value. New threads inherit copies of their parent’s parameter bindings, though the parameter values themselves are not copied. That is, when a child sets a parameter, it does not affect a parent; when it mutates the state of a

parameter, the change is globally visible. The server deals with only two of MrEd’s standard parameters: *current-custodian* and *exit-handler*. The default *exit-handler* halts the entire runtime system. Setting this parameter to another function can cause conditions that would normally exit to raise an exception or perform clean up operations.

Finally, MrEd provides a mechanism for managing resources, such as threads (with associated parameter bindings), Transmission Control Protocol (TCP) listeners, file ports, and so on. When a resource is allocated, it is placed in the care of the current *custodian*, the value of the *current-custodian* parameter. Figure 3.1 specifies the only relevant operation on custodians: *custodian-shutdown-all*. It accepts a custodian and reaps the associated resources: it kills the threads in its custody, closes the ports, reclaims the TCP listeners, and recursively shuts down all child custodians.

3.3 Serving Static Content

A basic Web server satisfies HTTP requests by reading Web pages from files. High-level languages ease the implementation of such a server, while retaining efficiency comparable to widely used servers. The first subsection explains the core of the server implementation. The second subsection compares performance figures of this server to Apache [8], a widely-used, commercially-deployed server.

3.3.1 Implementation of the Web Server’s Core

The core of a Web server is a connection loop. It listens for TCP connections to a particular port. When a client connects, the server creates a new thread to process the connection. Then the server recurs:²

```
;; server-loop : Tcp-listener → Void
(define (server-loop listener)
  (let-values ([ip op] (tcp-accept listener)))
    (thread (λ () (serve-connection ip op))))
  (server-loop listener))
```

²*let-values* binds names to the values returned by multiple-valued computations such as *tcp-accept*, which returns input and output ports.

For each request, the server parses the first line and the optional headers:

```
;; serve-connection : Input-port Output-port → Void
(define (serve-connection ip op)
  (let-values ([meth url-string major-version minor-version]
              (read-request ip op))]
    (let* ([headers (read-headers ip op)]
           [url (string→url url-string)]
           [host (find-host (url-host url) headers)])
      (dispatch meth host port url headers ip op))))

;; read-request : Input-port Output-port → * Symbol String String String
;; to read a request from ip, to parse it, and to determine the
;; request method (get, put), URL, and protocol versions
;; effect: raises an exception and closes the ports, if parsing fails
(define (read-request ip op) ...)
```

A dispatcher uses this information to find the correct file corresponding to the given URL. If it can find and open the file, the dispatcher writes the file's contents to the output port; otherwise it writes an error message. In either case, it closes the ports before returning.³

	Connections/Second								
	1kB file			10kB file			100kB file		
Clients	MrEd	Apache	Ratio	MrEd	Apache	Ratio	MrEd	Apache	Ratio
2	967.5	1557.9	62.1%	655.1	771.6	84.9%	105.2	113.2	92.9%
4	986.7	1623.4	60.8%	772.0	1084.4	71.2%	110.2	115.7	95.2%
8	997.9	1607.0	62.1%	752.9	1099.0	68.5%	116.0	115.8	100.2%
16	982.8	1597.0	61.5%	782.6	1101.3	71.1%	116.5	116.1	100.3%
32	923.8	1551.0	59.6%	760.7	1104.0	68.9%	116.7	116.3	100.3%
64	917.6	1577.2	58.2%	787.1	1093.0	72.0%	115.1	116.5	98.8%
128	946.3	1547.8	61.1%	769.4	1104.1	69.7%	116.7	116.5	100.2%

Ratio = MrEd/Apache

The client and server software each ran on an Advanced Micro Devices (AMD) Athlon 800MHz processor with 192 Mbytes of memory, running FreeBSD 4.1.1-STABLE, connected by a standard 100 Mbit/s Ethernet connection.

Figure 3.2: Performance for static content server

3.3.2 Performance

It is easy to write compact implementations of systems with high-level constructs, but they must not sacrifice performance for abstraction. More precisely, the server must serve content from files at about the same rate as Apache [8]. This goal seemed within reach because most of the

³The server actually handles multiple requests per connection. Section 3.4.5 discusses this further.

computational work involves parsing HTTP requests, reading data from disk, and copying bytes to a (network) port.⁴

To verify this conjecture, this section compares the server's performance to that of Apache on files of three different sizes. For each test, the client requested a single file repeatedly. This minimized the impact of disk speed; the underlying buffer cache should keep small files in memory. Requests for different files would even out the performance numbers according to Amdahl's law because the total response time would include an extra disk access component that would be similar for both servers.

The results in figure 3.2 show that the server essentially achieves the performance goals. The results were obtained using the S-client measuring technology [14]. For the historically most common case [11]—files between six and thirteen kB—the server performs at a rate of 60% to 80% of Apache. For larger files, which are now more common due to increased uses of multimedia documents, the two servers perform at the same rate. In particular, for one and ten kB files, more than four pending requests caused both servers to consume all available Central Processing Unit (CPU) cycles. For the larger 100 kB files, both servers drove the network card at full capacity.

3.4 Dynamic Content Generation

Over the past decade, the Web's content has become increasingly dynamic. *USA Today*, for instance, finds that as of the year 2000, more than half of the Web's content is generated dynamically [20]. Servers no longer retrieve plain files from disk but use auxiliary programs to generate a document in response to a request. These Web programs often interact with the user and with databases. This section explains how small changes to the code of section 3.3 accommodate dynamic extensions, that the performance of the revised server is superior to that of Apache,⁵ and that it supports a new programming paradigm that is highly useful in the

⁴This assumes that the server does not have a large in-memory cache for frequently-accessed documents.

⁵Apache outperforms most other servers for CGI-based content generation [5].

context of dynamic Web content generation.

3.4.1 Simple Scheme Servlets

Since a single server satisfies different requests with different servlets, the server dynamically invokes and links to servlets. More specifically, a Scheme servlet is a unit:

```
(unit/sig () (import servlet^) <def+exp> ... <exp>)
```

It exports nothing; it imports the names specified in the *servlet*[^] signature. The result of its final expression (and of the unit invocation) is a Hyper-Text Markup Language (HTML) page or some other type of document.

Servlets may produce responses of several variants. To support different media types, one response is a list of strings, where the first string is the Multipurpose Internet Mail Extensions (MIME) type [30] and the rest of the strings form the body of the HTTP message sent to the client. Responses also include a more specific variant convenient for HTML, a more general variant, and a variant suitable for streaming output.

The more general response type is a *response/full* built with the function

```
(make-response/full code message seconds mime extras body)
```

The *code* argument is an HTTP response code such as 202 for good responses, 301 for redirections, 404 for file not found, or 500 for internal errors. The *message* is a string describing the response *code*. The *seconds* number indicates the time the document was generated. The *mime* string encodes the MIME type. The *extras* association list maps HTTP header symbols to their string values. The *body* list of strings forms the contents of the response.

Streaming output is useful for responses that require a significant amount of time to either compute or transmit. Large audio or video media files benefit from streaming, as does the progress page for installing manuals in DrScheme's help-desk. The function *make-response/incremental* is the same as *make-response/full*, except the *body* argument is a function. The server calls the

<i>X-expression</i>	=	<i>String</i>
		(cons <i>Symbol</i> (cons (<i>listof</i> (<i>list</i> <i>Symbol</i> <i>String</i>)) (<i>listof</i> <i>X-expression</i>)))
		(cons <i>Symbol</i> (<i>listof</i> <i>X-expression</i>)) ;; an element with no attributes
		<i>Symbol</i> ;; symbolic entities such as
		<i>Number</i> ;; numeric entities like
		<i>Misc</i>

Figure 3.3: X-expression Responses

function with an output procedure that consumes a string. Each string contributes incrementally to the body of the HTTP response.

The most convenient response variant for HTML is an X-expression.⁶ The grammar in figure 3.3 describes this parenthesized version of the eXtensible Markup Language (XML) [19] in detail. The documentation for DrScheme’s XML collection describes the seldom used *Misc* elements.

Here is a trivial Scheme servlet⁷ using **quasiquote** [87] to create an X-expression HTML page with **unquote** (a comma) allowing references to the *TITLE* definition.

```
(require (lib "unitsig.ss")
         (lib "servlet-sig.ss" "web-server"))
(unit/sig () (import servlet^))
(define TITLE "My First Scheme Servlet")
‘(html (head (title ,TITLE))
      (body
        (p (center ,TITLE))
         (p "Hello, World!"))))
```

The script defines a title and produces a simple Web page containing a message.

The servlet’s imports supply the *initial-request*. A request contains the HTTP method, the URL, the optional headers, and the bindings:

```
method    : (union 'get 'post 'head)
url       : URL
headers   : (listof (cons Symbol String))
bindings  : (listof (cons Symbol String))
```

To extend the server with Scheme servlets, the *dispatch* function from section 3.3 redirects requests for URLs starting with `"/servlets/`. More concretely, instead of responding with the

⁶X-expressions appeared in an earlier work [71].

⁷All subsequent servlets in this dissertation will elide the **require** term.

contents of a file, *dispatch* loads a unit from the specified location in the file system. Before invoking the unit, the function installs a new *current-custodian* and a new *exit-handler* via a

parameterize expression:

```
;; in dispatch:
...
(if (servlet-url? url)
  (let ([cust (make-custodian)])
    (parameterize
      ([current-custodian cust]
       [exit-handler ( $\lambda$  (x) (custodian-shutdown-all cust))])
      (let ([a-servlet (cached-load (url-path url))]
            (output-xhtml (invoke-unit/sig a-servlet servlet^))))
    ...))
```

The newly installed custodian is shut down on termination of the servlet. This halts child threads, closes ports, and reaps the script's resources. The new *exit-handler* is necessary so that erroneous content generators shut down only the custodian instead of the entire server.

3.4.2 Content Generators are First-Class Values

Since units are first-class values in MrEd, the server can store content generators in a cache. Introducing a cache avoids some I/O overhead but, more importantly, it introduces new programming capabilities. In particular, a content generator can now maintain local state across invocations. Here is an example:

```
(let ([count 0])
  (unit/sig () (import servlet^))
  (set! count (add1 count))
  '(html (head (title "Testing Persistent State of Counter"))
    (body (p "This is a servlet generated Web page." )
           (p "The current count is " ,(number→string count))))))
```

This generator maintains a local count that is incremented each time the unit is invoked to satisfy an HTTP request. Its output is an HTML page that contains the current value of *count*.

3.4.3 Exchanging Values between Content Generators

In addition to maintaining persistent state across invocations, content generators may also need to interact with each other. Conventional servers force server programs to communicate via

the file system or other mechanisms based on character streams. This requires marshaling and unmarshaling data, a complex and error prone process. In the Scheme server's architecture, servlets can naturally exchange high-level forms of data through the common heap. The server permits multiple servlets to be defined, using compound-units, within a single lexical scope. This permits arbitrary code to define and initialize data (such as semaphores) and operations (such as useful shared abstractions) within this common scope.

3.4.4 Interactive Servlets

Christian Queinnec suggested [91] that Web browsing in the presence of dynamic Web content can be understood as the process of capturing and resuming continuations.⁸ For example, a user can bookmark a generated page that contains a form and (try to) complete it several times. This action corresponds to the resumption of the servlet's computation after the generation of the first form.

Ordinarily, programming this style of computation is a complex task. Each time the computation requires interaction (responses to queries, inspection of intermediate results, and so forth) from the user, the programmer must split the program into two fragments. The first generates the request for interaction, typically as a form whose processor is the remainder of the computation. The first program must store its intermediate results externally so the second program can access and use them. The staging and marshaling are cumbersome, error-prone, slow, and inhibit the reuse of existing non-Web programs that are being refitted with Web-based interfaces. To support this common programming paradigm, the server links servlets to the additional primitives in figure 3.4.

The *send/suspend* function sends an HTML form to the client for further input. The function captures the continuation and suspends the servlet's computation. When the user responds, the server resumes the continuation with the new request.

To implement this functionality, *send/suspend* consumes a function of one argument. This

⁸This idea also appears in Hughes's paper on arrows [64].

<i>send/suspend</i>	:	$(Url \rightarrow Response) \rightarrow Request$
<i>send/forward</i>	:	$(Url \rightarrow Response) \rightarrow Request$
<i>send/back</i>	:	$Response \rightarrow Void$
<i>send/finish</i>	:	$Response \rightarrow Void$
<i>adjust-timeout!</i>	:	$Number \rightarrow Void$

Figure 3.4: Additional content generator primitives

function, in turn, consumes a unique URL and generates a form whose action attribute refers to the URL. When the user submits the form, the suspended continuation is resumed. Consider figure 3.5, which presents a simple example of an interactive content generator. This script implements carried multiplication, asking for one number with one HTML page at a time. The two underlined expressions represent the intermediate stops where the script displays a page and waits for the next set of user inputs. Once both sets of inputs are available it produces a page with the product.

In general, this paradigm produces programs that naturally correspond to the flow of information between client and server. These programs are easier to match to specifications, to validate, and to maintain. The paradigm also causes problems, however. The first problem, as Queinnec points out, concerns garbage collection of the suspended continuations. By invoking *send/suspend*, a servlet hands out a reference to its current continuation. Although these references to continuations are symbolic links in the form of unique URLs, they are nevertheless references to values in the server's heap. Without further restrictions, garbage collection cannot be based on reachability. To make matters worse, these continuations also hold on to resources, such as open files or TCP connections, which the server may need for other programs.

Giving the user the flexibility to bookmark intermediate continuations and explore various choices creates another problem. Once the program finishes interacting with the user, it records the final outcome by updating persistent state on the server. These updates must happen at most once to prevent catastrophes such as double billing or shipping too many items.

```

(unit/sig () (import servlet^ )

;; get-number : String → (String → Html-page)
(define (get-number which-one)
  (let ([bindings
        (request-bindings
         (send/suspend
          (λ (k-url-identifier)
            `(html (head (title ,which-one " number"))
                  (body
                   (form ((method "post") (action ,k-url-identifier))
                        "Enter the " ,which-one " number:" nbsp
                        (input ((type "text") (name ,which-one)))
                        (input ((type "submit") (name "submit")))))))))]
    (extract which-one bindings)))

;; string-multiply : String String → String
...

;; extract : String (listof (cons Symbol String)) → String
...

(html (head (title "Product"))
      (body (p "The product is: "
              ,(string-multiply (get-number "first")
                                (get-number "second"))))))

```

Figure 3.5: An interactive servlet

Based on this analysis, the server implements the following policy. When the servlet calls the *send/finish* primitive, the server releases all the continuations associated with the servlet's computation. Servlets that wish to keep the continuations active can call *send/back* instead. When a user attempts to access a reclaimed continuation, a page directs them to restart the computation. Servlets permit the maximal amount of backtracking by delaying all of the updates to external persistent state until the last interaction. If a servlet must prevent a consumer from backtracking during the middle of a session, the *send/forward* primitive will clear the servlet's suspended continuations before inserting a continuation for the current interaction. This enables consumers to proceed forward, but not return to previous interaction points. Figure 3.6 compares the navigational capabilities of each interaction primitive.

These continuation reaping interaction primitives would have prevented several errors en-

countered in practice if commercial developers used them. A professor at Brown University encountered this type of error when renewing two Internet domain name registrations. The penultimate page of the registration program indicated that the user should wait for the server to finish processing the renewal request. After a moment, it automatically proceeded to the final page, confirmed the renewal, and billed the author's credit card. Accidentally hitting the back button returned to the processing page, which billed the credit card again, renewing the domain names for a second year. I also encountered a similar experience when saving a receipt page. Saving reloaded the uncached page, which resulted in a double billing error.

primitive	resumable	backtrackable
<i>send/finish</i>	no	no
<i>send/back</i>	no	yes
<i>send/forward</i>	yes	no
<i>send/suspend</i>	yes	yes

Figure 3.6: Comparison of Interaction Primitives

Furthermore, each instance of a servlet has a predetermined lifetime after which continuations are disposed. Each use of a continuation updates this lifetime. A running continuation may also change this amount by calling the *adjust-timeout!* primitive. This mechanism for shutting down the servlet only works because the reaper and the servlet's thread share the same custodian. This illustrates why custodians, or resource management in general, should not be identified with individual threads.

3.4.5 Implementing Interactive Servlets

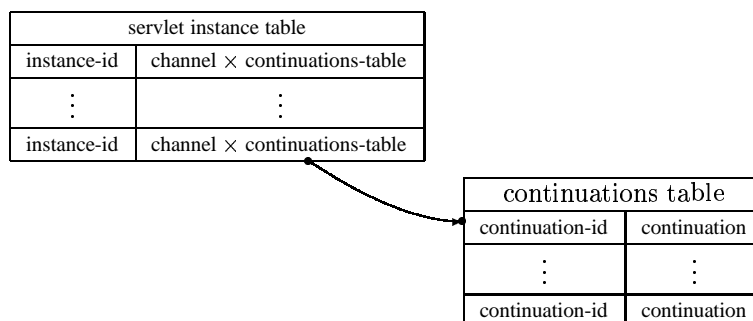
The implementation of the interactive servlet policy is complicated by MrEd's restriction that capturing a continuation is local to a thread and that a continuation can only be resumed by its original thread. To comply with this restriction, *send/suspend* captures a continuation, stores it in a table indexed by the current servlet, and causes the thread to wait for input on a channel. When a new request shows up on the channel, the thread looks up the matching continuation

in its table and resumes it with the request information in an appropriate manner. A typical continuation URL looks like this:

```
http://www/servlets/send-test.ss?id38*k3-839800468
```

This URL has two principle pieces of information for resuming computation: the thread (`id38`) and the continuation itself (`k3`). The random number at the end serves as a password preventing other users from guessing continuation URLs.

The table for managing the continuations associated with a servlet actually has two tiers. The first tier associates instance identifiers for servlets with a channel and a continuation table. This continuation table associates continuation identifiers with continuations. Here is a rough sketch:



When a thread processes a request to resume its servlet's continuation, it looks up the servlet in one table, and extracts the channel and continuation table for that servlet. The server then looks up the desired continuation in this second table and passes it along with the request information and a channel to receive the response.

The two-tier structure of the tables also facilitates clean-up. When the time limit for a servlet instance expires or when its computation terminates, the instance is removed from the servlet instance table. This step, in turn, makes the continuation instance table inaccessible and thus available for garbage collection. The two-tiered structure also facilitates implementing *send/finish*.

Interactive servlets and network connections that handle multiple requests complicate resource management. Servlets and their resources survive across multiple connections. Connec-

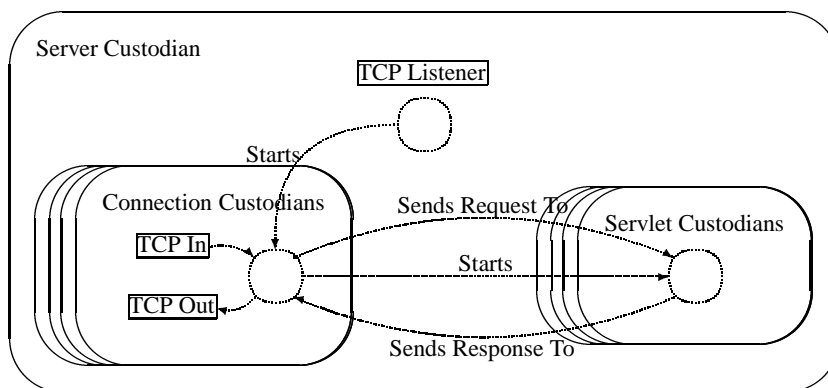


Figure 3.7: Server Resources

tions direct requests to multiple servlets. Thus, the birth and death of connections and servlets proceed independently. If servlets access the network connection directly, several problems result. First, handling multiple requests during a single network connection requires that the server and client distinguish the boundaries between the multiple requests and responses. A defective servlet could transmit erroneous data that prevents the accurate detection of these boundaries.⁹ Second, if a browser drops the network connection¹⁰ the servlet should not fail; rather the connection should.

A more functional programming style and MrEd's custodians meet this challenge nicely. Figure 3.7, depicts the layout of custodians within the Web server. The entire server runs within a single, top-level custodian, so the administrator can easily shutdown the entire server. Each time the server listens for a new TCP connection on the HTTP port, the server first creates a new custodian to manage the connection. After creating new I/O ports for the connection, the server loop starts a new thread within the connection custodian to process the connection. That is, even though there is a one-to-one correspondence between threads and custodians, the parent thread allocates new network ports within the child's custodian before the child thread

⁹Omitting the Content-length header without using the Chunked content encoding creates this problem.

¹⁰The consumer may click the Stop button. A roommate could break a dial-up connection by attempting to use the phone. Power failures, crashing browsers, or any number of natural or artificial disasters can suddenly terminate network connections.

exists. Thus, the separation of threads and custodians helps with the situation.

Requests for new servlet sessions cause the connection thread to create a new servlet custodian. Instead of imperatively reading from and writing to the network connection as the servlet runs, the servlet receives Web requests from the connection thread via a channel. The servlet constructs a response value, which it sends back to the connection thread via another channel. This leaves all of the imperative I/O and network anomalies to the connection and the interesting computation to the servlet.

3.4.6 Performance

The Scheme Web server ought to outperform conventional servers that execute ordinary CGI scripts. A conventional server starts a separate OS process for each incoming request, creating a new address space, loading code, etc. The server eliminates these costs by avoiding the use of OS process boundaries and by caching servlets.

These experiments confirm that the server handles more connections per second than CGI programs written in C. For example, figure 3.8 clocks a C program's binary in CGI and FastCGI against a Scheme script producing the same data. The table does not contain performance figures for responses of 100 kB and larger because for those sizes the network bandwidth becomes the dominant factor just as with static files.

The table also indicates that both the standard CGI implementation and the Scheme servlets scale much better relative to response size than FastCGI does. It seems likely that this results from FastCGI copying the response twice, and is thus much more sensitive to the response size. Of course, as computations become more intensive, the comparison becomes one of compilers and interpreters rather than of servers and their protocols.

3.4.7 Modularity of the Server

Web servers must not only be able to load Web programs (e.g., CGI scripts or servlets) but also load new modules in order to extend their capabilities. For example, requiring password

Clients	CGI		FastCGI		MrEd Full		MrEd Lite	
	1kB	10kB	1kB	10kB	1kB	10kB	1kB	10kB
8	161.1	158.7	742.7	551.6	766.5	665.9	851.4	742.6
16	157.6	156.9	728.8	547.2	759.6	659.3	847.3	727.9
32	153.4	153.1	720.7	544.4	733.8	627.4	837.8	721.4

MrEd Full is the server described in this chapter. It includes the continuation reaper described at the end of section 3.4.4, whose implementation is currently quite inefficient. MrEd Lite disables this reaper (rendering *send/suspend* less usable), making its services more directly comparable to those of CGI and FastCGI.

Figure 3.8: Performance for dynamic content generation

authentication to access particular URLs affects serving content from files and from all servlets. In order to facilitate billing various groups hosted by the server, the administrator may find it helpful to produce separate log files for each client instead of a monolithic one. A flexibly structured server will split key tasks into separate modules, which can be replaced at link time with alternate implementations.

Apache's module system [107] allows the builder of the Web server to replace pieces of the server's response process, such as those outlined above, with with their own code. The builder installs structures with function pointers into a Chain of Command pattern [53]. Using this pattern provides the necessary extensibility but it imposes a complex protocol on the extension programmer, and it fails to provide static guarantees about program composition.

In contrast, the server was originally constructed in a completely modular fashion using the unit system [47]. This provided the flexibility of the Apache module system in a less ad hoc, more hierarchical manner. To replace part of how the server responded to requests, the server builder wrote a compound unit that linked the provided units and the replacement units together, forming an extended server. Naturally, the replacement units may have linked to the original units and delegate to them as desired.

Using units instead of dynamic protocols had several benefits. First, the server didn't need to traverse chains of structures, checking for a module that wants to handle the request. Second, the newly linked server and all the replacement units were subject to the same level of static

analysis as the original server. (DrScheme's soft-typing tool [46, 75] revealed several easily corrected premature end-of-file bugs in the original server.)

Chapter 4

Compiling to the Web

4.1 Designing Web Programs

The proceeding chapters show the obvious need for generating Web information on demand. One Web page may need the current time and date; another page may include results from a database query; a third page may display the current status of the server. Since such programs compute small amounts of information and produce not much more than a single Web page, people call them *scripts*.

Following a long-standing tradition in computing, Web scripting has grown up. These scripts have now turned into serious, maintained programs that sometimes represent the *raison d'être* of a commercial establishment. Consumers can find on-line stores, e-mail clients, interactive games, and more implemented with a Web interface. In other words, instead of writing Web *scripts*, programmers now design, implement, and maintain interactive Web *programs* with complex and multi-layered interface protocols. Thus, all the usual software engineering concerns about evolving maintainable code to match growing requirement specifications apply.

Furthermore, the designers of complex, interactive server-side Web programs face an additional software engineering problem when using existing technology. Most dialogs consist of many interactions, where each interaction presents a form and processes the user's response. However, CGI programs halt after processing a single form. Similarly, Java servlet `doGet` or `doPost` methods return upon handling inputs from a single form. Java Server Pages also force

programmers to contort their code to respond with a single page in response to a single interaction with the user. Since all widely used Web technologies suffer from the same problem of forgetting control information between interactions, the rest of this chapter's discussions of CGI programs applies equally well to other standards.

To force the interactive nature of programs into the Web programming mold, an interaction is implemented by having a script deliver a Web page, wait for the consumer to submit a response, and then process that response with a(nother) script. Complicating matters even more, the Web programs must accommodate consumers who backtrack in their interactions, clone their browser windows, re-submit the same or different answers for any given form, and so on. In short, a Web program and a consumer make up a pair of coroutines where each interaction point can be resumed *arbitrarily often*. However, due to the lack of these multiply-resumable coroutines or similar constructs in most Web programming languages, the designer cannot match the structure of the interaction with the structure of the program. These requirements result in ad hoc mechanisms to save and restore control state that are difficult to develop, maintain, or explain to colleagues.

Chapter 3 shows that Web programmers can use *existing* software engineering methods to develop interactive programs, while this chapter shows how well-known, automatable transformations can *generate* standard CGI scripts from these programs. Specifically, extending a programming language with a primitive for Web interactions simplifies the design, development, and maintenance of interactive Web programs; how it allows programmers to migrate legacy programs to the Web; how the resulting programs manage the two kinds of information flows found in Web programs; and how to adapt existing programming environments in support of this development style.

The remainder of this chapter is organized as follows. The second section briefly introduces conventional Web programming. The third section presents the central ideas of the chapter: an alternative implementation of Web interaction constructs. The fourth section illustrates that

when Web programs are just interactive programs, programmers can develop, test, and debug them in ordinary programming environments, enriched with a small run-time extension. The fifth section outlines an implementation of these ideas in Scheme, that enables the testing of each development stage. The work does not rely on Scheme's advanced control constructs, however.

4.2 Interactive CGI Programs

A typical interactive program performs a series of computations interspersed with interactions with the user. Each interaction requests information using HTTP's GET or POST methods [42] and waits for the user's response. After the last interaction, the program produces the final result. This section demonstrates how programmers port interactive applications to existing Web technologies, first via conventional means and then in a more direct manner.

4.2.1 Conventional CGI Programs

Figure 4.1 presents a trivial interactive Scheme program that requests two numbers, adds them, and displays the result. The footnoted boxes only exist for explanation purposes; they are not part of the program text. Converting even this simple program to function as a Web script complicates the code tremendously. According to the CGI standard, every time the program sends an HTML form to the consumer's browser, the CGI program terminates. When the user submits a response to the form, the server starts the CGI script that the form specified as its processor. That is, if an interactive program contains a *single* input request, its equivalent CGI script consists of two separate fragments. The problem is, however, even more complex than that because the consumer may use the back-button to return to a page and may re-submit the same or different answers. Worse, using the new-window functionality to clone a browser, the consumer can submit two responses to a single form (more or less) simultaneously.

To accommodate these uses, a programmer must—at least conceptually—turn an interactive program into a coroutine; the consumer plays the role of the second coroutine. One way to

```

;; prompt-read : String → Value
;; read a Scheme value
(define (prompt-read question) ;; defines the function prompt-read
  (display question)
  (read))

;; main
(display)
(+ (prompt-read "Enter the first number to add:")
   (prompt-read "Enter the second number to add:"))

```

Figure 4.1: An Interactive Addition Program

```

;; produce-html : String String (listof Value) → void
;; effect: to write a CGI HTTP header and HTML Web form
(define (produce-html question mark free-values) ...) ;; body uninteresting

(define FIRST-STOP "first number done")
(define SECOND-STOP "second number done")

(define bindings (get-bindings))

;; main
(cons ;; each bracketed clause is a question-answer pair;
      ;; in this instance, all answer expressions are boxed
      [(empty-bindings? bindings)
       (produce-html "Enter the first number to add:" FIRST-STOP '())]
      [(string=? (extract-binding/single 'continue-at bindings) FIRST-STOP)
       (produce-html "Enter the second number to add: " SECOND-STOP
                   (list (list 'first-number (extract-binding/single 'response bindings))))]
      [(string=? (extract-binding/single 'continue-at bindings) SECOND-STOP)
       (display (+ (string→number (extract-binding/single 'first-number bindings))
                  (string→number (extract-binding/single 'response bindings))))])

```

Figure 4.2: A CGI Version of Figure 4.1

accomplish this is to separate the program into several fragments, one per interaction and one for the last step. When a fragment has finished its task, the execution stops. All information from one program fragment required by some later fragment must be communicated explicitly. All the methods for communicating with the next fragment marshal the data into a string and

transmit it in a hidden HTML field, in a cookie, or save it in a file on the server.

Figure 4.2 shows the addition program converted into a CGI program. Because the original addition program contains two interactions, the corresponding CGI version consists of three fragments, re-integrated into a single program via a conditional. The invocation of *get-bindings* extracts the bindings from the Web form, which the CGI program then tests for three conditions:

1. If there are no bindings, the program starts from the beginning. It creates a Web page with a question, a hidden field that specifies the resumption point, and the list of values that are supposed to be hidden in the Web page.
2. If the program can extract *FIRST-STOP* for 'resume-at, then it was invoked with a first input. It produces a second form and queries the consumer for another number.
3. Finally, if the program extracts *SECOND-STOP* for 'resume-at, it has obtained both numbers and can produce the sum.

As the computation unfolds, all necessary values are passed explicitly from one stage to the next as in a bucket brigade.

Clearly, the structure of the CGI program radically differs from that of the original version—indeed, it is basically inverted¹—yet their behavior per se is identical. The inverted structure of the second program is necessary because of the constraints of the CGI standard and the capabilities of the browsers. In particular, a consumer can create a “curried adder”² using the back button to re-enter different values for the second argument. The situation only grows more grim as the number of interactions increases. In general, the program may loop, requesting an arbitrary number of inputs. This necessitates constructing a single branch that handles many responses, remembering the state of the iteration and an unbounded number of intermediate values. Performing this restructuring manually easily leads to errors.

¹M. Jackson [66] recognized a similar structural problem in the early 1970s. When COBOL programs consume tree-shaped data in one file and produce a different tree-shaped form of data in another file, it is best to think of the program as two coroutines. Since COBOL doesn't support coroutines, he invented *program inversion*, a technique for providing simple coroutine-like procedures in languages that don't support such forms of control.

²A curried function accepts some prefix of its arguments and returns a new function that accepts the remaining arguments.

In principle, the CGI programs are systematically related to the “direct style” interactive programs that use plain input and output primitives. While CGI programmers currently structure each script independently, the software construction process should take advantage of this relationship. The next sections demonstrate how to automatically transform a direct-style program into a CGI program, with no intervention from the programmer

4.2.2 Direct-Style CGI Programs

Software engineers have learned how to develop and maintain sequential interactive programs. Hence, if they could develop interactive programs and use them as CGI scripts, they could reuse the software engineering techniques for interactive programs in this chaotic world of Web programming.

Since CGI programs run in the context of a Web server, a custom server can provide CGI programs with re-implementations of primitives such as *display* or *prompt-read*. A specialized version of *prompt-read* can capture the current control state as a continuation [103, 88] value, using Scheme’s *call/cc* construct. The server can store this continuation for later resumption. The server associates the continuation with a new URL that accepts the inputs from a Web form. When the consumer submits a response to this Web form, the browser issues a request for the URL that is associated with a continuation. This request and all future requests for the URL resume the continuation with the data from the Web form. In particular, because a Scheme continuation can be invoked an arbitrary number of times, the consumer can respond to the same Web form a multiple number of times and thus resume a continuation as often as desired.

Chapter 3 implements this approach and demonstrates its advantages. In addition to facilitating program construction, the modified Web server yields superior speed for CGI scripts compared to several existing methods.

Unfortunately, the approach has two problems in theory. First, it requires a server written in a language with advanced control features such as continuations. Second, the URLs for

continuations act as persistent references to storage within the server. This results in a distributed garbage collection problem with no support from the browser. In fact URLs may be in bookmark files, human minds, and other media. One way to address this problem is to impose timeouts. That is, the server disposes of unused continuations after some given amount of time. Unfortunately, time-outs provide a less than ideal solution. If a timeout is too large, the server consumes too much memory. If it is too short, it forces consumers to restart computations from the beginning too often. It also makes the consumer depend on the reliability of the server, which may restart due to power failures or software upgrades.

Several months of actual experience using the server for an outreach project's Web sites [1, 2] revealed that problems with timeouts matter in practice.³

- One of the sites contained a workshop registration form with a timeout of 24 hours. This sufficed for most respondents; a few, however, had to request an extension due to a snow-storm that interfered with their Internet access. Unfortunately, not even the site operator can resurrect a continuation that the server has discarded.
- On another occasion, one of the authors copied the first page generated by the registration program to a different file. Initial testing suggested that the copied page functioned correctly, yet a few days later several workshop administrators indicated otherwise. Even though neither the code nor the static pages changed, the form ceased to function since the continuation had timed out.

4.3 Generating CGI Programs

The theoretical and practical problems with the server-based approach suggests searching for an alternative implementation technique. This section describes this new approach, first for purely functional programs, and then for programs utilizing a mutable store.

³Also, because the generated URLs encode enough information to identify the instance of the program, its continuation, and a random key, they are too long for some email clients, which mangled them. Some users reported problems copying the URLs because of this.

```

(define-struct closure (code env))
;; Closure = (make-closure Int Env)
;; Env = (listof Value)

;; apply-closure : Closure (listof Value) * → Value
(define (apply-closure f . args)
  (apply ;; supplies the arguments
         (apply ;; supplies the environment
              (vector-ref closures (closure-code f))
              (closure-env f))
         args))

;; the converted functions and continuations
(define closures
  (vector
   (λ () ;; the environment (in this case, empty)
     (λ (response1) ;; the argument
       (prompt-read-k "Enter the second number to add:" (make-closure 1 (list response1))))

     (λ (response1) ;; the environment (in this case, holds the previous argument)
       (λ (response2) ;; the argument
         (display (+ response1 response2))))))

  ;; prompt-read-k : String Closure → void
  (define (prompt-read-k s k)
    (display s)
    (apply-closure k (read)))

  ;; main
  (prompt-read-k "Enter the first number to add:" (make-closure 0 empty))

```

Figure 4.3: The Compiled Version of Figure 4.1

4.3.1 Functional CGI Programs

Removing timeouts would eliminate many of the problems encountered with the custom Web server. Since timeouts reclaim resources on the server consumed by suspended continuations, an alternate implementation that saved control state on the client would render timeouts unnecessary. Techniques for compiling functional programming languages eliminate the uses of *call/cc* to suspend continuations. More specifically, three well-known transformations automatically create the control flow required for Web applications:

Continuation Passing Style (CPS) eliminates *call/cc* by representing the control state of a program explicitly. In particular, each function of the program now consumes one additional

argument: another function representing the continuation. A function that must grab the continuation and store it for future use can simply refer to this new argument. In this case, a re-implementation of *prompt-read* can turn its new argument into a resumption point, that is, a point from where the program can be restarted.

Lambda lifting [68] turns the resumption points into independent functions that can be moved to the top level, making them accessible to the code handling the next interaction.

Defunctionalization [94] changes the representation of higher-order data, such as closures⁴ and continuations, into a first-order form. By choosing portable concrete representations (in this case, vectors), the runtime system can correctly marshal these kinds of higher-order data. Using defunctionalization, the script writes the continuation into a hidden field of a Web form and uses it later to restart its computation.

CPS'ing, lambda lifting, and defunctionalizing partitions a program into separate interactive steps, so computation can halt conveniently between them. Small changes then convert the program into a standard CGI script.

A trivial but illustrative example from figure 4.1 explains the process. The result of these three automated translation steps is shown in figure 4.3. This interactive program requires one final step to become a CGI program. The revision in figure 4.4 demonstrates the result of systematically transforming the compiled version into a CGI script. The result is structurally almost identical to the hand-coded version of figure 4.2.

The details of the process are as follows. The first step produces a CPS'ed version of the program. Here is a running example:

```
(prompt-read-k "Enter ... fi rst ... "
;; λ declares anonymous, first-class functions
(λ (res1)
  (prompt-read-k "Enter ... second ...:"
    (λ (res2)
      (display (+ res1 res2))))))
```

⁴Closures are functions that remember the lexical context of their creation. They usually consist of an environment and a code pointer.

```

(define-struct closure (code env))
;; Closure = (make-closure Int Env)
;; Env = (listof Value)

;; apply-closure : Closure (listof Value) * → Value
(define apply-closure ...); as in figure 4.3

(define closures ...); as in figure 4.3

;; replaced:
(define (prompt-read-k s k)
  (produce-html s (closure-code k) (closure-env k)))

;; added:
;; produce-html : String String (listof Value) → void
;; effect: to write a CGI HTTP header and HTML Web form
(define (produce-html question mark free-values)
  ...)

(define bindings (get-bindings))

;; main
(cond
 [(empty-bindings? bindings)
  (prompt-read-k "Enter the first number to add: " (make-closure 0 empty))]
 [(string=? (extract-bindings/single 'resume-at bindings) "0")
  (apply-closure (make-closure 0 (create-env-from-strings (extract-bindings/single 'env bindings)))
    (extract-binding/single 'response bindings))]
 [(string=? (extract-bindings/single 'resume-at bindings) "1")
  (apply-closure (make-closure 1 (create-env-from-strings (extract-bindings/single 'env bindings)))
    (extract-binding/single 'response bindings))])

```

Figure 4.4: The CGI Version of Figure 4.3 (Compare with Figure 4.2)

where

```

;; prompt-read-k :
;; String (Value → Value) → Value
(define (prompt-read-k s k)
  (display s)
  (k (read)))

```

The CPS converter must supply alternate implementations of primitives. CPS'ed versions of higher-order primitives that accept (or return) call-backs must supply a continuation to their argument, which may after all contain resumption points. External modules that accept function arguments must be transformed as well.

Lambda lifting turns anonymous functions into globally defined functions. It thus allows the compiled CGI program to resume a continuation with a call to a global function. Each expression of the form

```
(λ ⟨args⟩ ⟨body⟩ ...)
```

is replaced with

```
((λ ⟨free-vars⟩
  (λ ⟨args⟩ ⟨body⟩ ...))
  ⟨free-vars⟩)
```

where $\langle free\text{-}vars \rangle$ is the list of free variables in $\langle body \rangle \dots$. This new function is closed, so it can be safely lifted to the outermost lexical scope.

For this chapter's running example, this step yields the code in figure 4.5. The functions *closure1* and *closure2* enable the program to start from different resumption points, turning the original program into a curried adder just as the back button on a Web browser does.

```
(def ne closure1
  (λ ()
    (λ (res1)
      (prompt-read-k "Enter ... second ...:"
        (closure2 res1))))))

(def ne closure2
  (λ (res1)
    (λ (res2)
      (display (+ res1 res2))))))

(prompt-read-k "Enter ... first ...:" (closure1))
```

Figure 4.5: Lambda Lifted

Figure 4.3 shows the result of the final compilation step, namely of converting closures into structures; function applications are performed by *apply-closure*. The step is necessary for two reasons. First, Web forms must refer to a specific resumption point (closure) within a program, but Web forms can only contain strings. A unique symbolic code, such as an index into a vector of closures, satisfies this requirement. Second, some closures may survive an interaction with the consumer, which means that their environment must be marshaled into strings for hidden

fields and unmarshalled upon resumption. Since all closures have been converted into first-order *closure* structures, a function such as *prompt-read* can write a closure into the hidden field of a Web form and the CGI program can read this closure and apply it. Specifically, the code pointer of the continuation describes what subprogram to invoke next. The continuation's environment captures any values needed by the next subprogram instead of explicitly passing them in hidden fields.

Up to this point, the transformation produced a semantically equivalent program, so the result is a normal interactive program. Replacing two fragments of the defunctionalized program produces a CGI program. The definition of *prompt-read* changes and now marshals the continuation into a Web form, prompts the user with a form, and then exits. The main program changes to the text of figure 4.4. In other words, the program first checks the form bindings for the continuation from *prompt-read*. If it exists, the continuation is resumed via a closure application. If not, the invocation starts from the beginning.

Proving the well-behaved nature of this transformation, requires a modified notion of observational equivalence that accounts for the differences in the two programs. Intuitively, disallowing the client's use of the back button, cloning, and bookmarking facilities would force each continuation resumed to be the last one suspended, thus maintaining the same control flow as the original interactive program. More formally, this notion of equivalence would restrict the contexts used for observations to only include streams of inputs where each continuation in the stream must match the one produced from processing the stream up to that point. Since each transformation step preserves either full or restricted observational equivalence, the entire process would preserve the restricted form of equivalence.

Security

Recording the continuation in the client and retrieving it introduces two security issues. First, malicious users can alter the continuation, resulting in unexpected behavior. Second, curious

users can inspect the continuation’s free variables, possibly revealing confidential information.

Existing cryptographic solutions remedy both these problems without introducing more than a fixed amount of server-side state. Appending the marshalled continuation with a keyed hash [4] would allow the unmarshaller on the server to verify the continuation’s integrity. Encrypting the continuation using a block cipher with a random key kept only on the server would prevent users from inspecting the continuation. The system could generate the necessary keys on a system wide or per-program basis, avoiding excess server-side state. One mode of the proposed Advanced Encryption Standard [32] simultaneously does block encryption as well as message authentication in one (highly parallelizable) operation.

4.3.2 Compiling Stateful CGI Programs

While generating CGI programs from interactive functional programs is almost a routine task with functional compilation techniques, *internal*⁵ assignments in the interactive program pose an interesting challenge. The first problem is due to plain variable assignments—**set!** in Scheme—because lambda lifting assumes that copying bindings is acceptable. The compiler must therefore eliminate all variable assignments with a transformation that replaces mutable variables by boxes,⁶ assignments to variables with assignments to boxes, and references to such variables with dereferences of boxes. Furthermore, the CGI program generator must know all boxes that the original program uses (or implicitly introduces). Figure 4.6 contains an imperative version of the running example converted to use Scheme boxes.

The second problem is much more severe. Semantically, assignments introduce an additional element: the store. Roughly speaking, the store is threaded through the program, independently of the control state. In particular, when a Scheme program invokes the same continuation twice, the store of the second invocation reflects all the store updates since the first invocation. Modifications of the store survive continuation capture and invocation.

⁵Modifications of data in *external* entities, say the server file system or a database, are already well-understood.

⁶Boxes in Scheme are akin to wrapper classes in Java.

A consumer who invokes the same continuation twice via a Web form should also see that the store modifications of the first invocation survive when the second invocation is launched. This requirement implies that a CGI program must deal with the store differently than with the environment of a closure. In particular, it is wrong to place the current store into a hidden field of a Web form. After all, if the consumer cloned the page, the browser would also copy the store, and two submissions of the form would submit the same store twice.

Still, the system must remember the current store somewhere when a CGI program suspends. It could either place the store on the server or on the client machine. The same arguments regarding the placement of continuations also indicate that the server is ill-suited for this purpose.⁷ Hence, the store must become a datum that is sent to, and then stored on, the consumer's machine—but not inside the Web page.

This reasoning leaves the single choice of turning the store into a browser “cookie” and placing this marshalled form into the consumer's cookie file. Unlike hidden fields, they are independent of any particular page, so changing continuations via the back button does not affect the store. Figure 4.7 sketches the cookie-based translation of figure 4.6.

Although this naïve cookie solution sounds straightforward, it has two imperfections. The first one, which is minor, is a the restriction that Web browsers have a limit of 80kB of storage for cookies per host name [72]. In principle, a limit like this is no different than a limit on heap space for a conventional program, but the small size of the limit will be problematic for some programs. As security research improves, cookies or some other mechanism may mature enough to lift these simplistic restrictions. The second, more important, problem arises because browsers transmit cookies at the time they submit the Web request. If the user submits simultaneous requests, the second request processed by the server will contain an outdated cookie. A naïve implementation may thus lose updates to the store.

One solution is to include a sequence number [92] with the cookie store. A sequence number

⁷Avoiding server-side state also facilitates replicating the server across several machines. Although outside the scope of this dissertation, replication improves industrial servers' load balancing and fault resistance.

allows the CGI program to detect race conditions. More specifically, the CGI stub code stores a sequence number for each original invocation (“session”) of a CGI program and uses this sequence number to manage access to the store. If it ever obtains a store with a sequence number less than the current one, it asks the consumer to resubmit the Web form. Unfortunately, the use of sequence numbers re-introduces the server side storage management problem, yet because the storage needs for numbers are small, the problem is probably negligible.

In summary, the inventors of browsers created two mechanisms for threading information through Web computations. The two mechanisms are analogous to the two ways information flows in a programming language semantics: stores that accumulate over time and continuations with environments that grow and shrink. The CGI compiler can therefore use the browsers’ mechanisms to implement the separate storage requirements for continuations and stores in a systematic manner.

4.4 Developing CGI Scripts

Developing a conventional CGI program in standard programming environments is difficult. To debug the program properly, the developer should run the program as a CGI script and interact with it through a browser. This is, however, a poor interaction environment. Instead of a proper error message, the programmer sees responses such as

```
Internal Server Error...More information about this error may  
be available in the server error log.
```

The server’s error log contains a corresponding report:

```
Premature end of script headers
```

followed by the name of the program. The programmer can infer from this that the CGI program didn’t output a valid response before terminating, but little more.

Any compilation process introduces the additional problem that the code that is executed as a CGI script is not the direct-style code that the programmer wrote. Instead, the programmer's code is first transformed and then run under the server's control.

Thankfully, an extension of existing programming environments overcomes both problems. The idea is to provide a library that re-implements primitives such as *prompt-read* so that the execution of the direct-style program functions as if the CGI script were run. In particular, the primitive communicates the given Web page to a browser, and the browser communicates the submission of a Web form to these primitives. Furthermore, the new library keeps track of the continuations of *prompt-read* so that the developer can truly simulate a consumer's actions on the browser.

A library (technically, a Teachpack [44]) demonstrates this idea. The library provides interaction functions for the DrScheme programming environment [44] which produce Web forms and return the values of the submitted fields. The re-implemented *prompt-read* primitive uses a more general primitive that accepts HTML pages (with forms); it grabs the current continuation, stores it, and manages the communication with the browser. By switching Teachpacks, legacy software can run either as a command line program or as a Web application.

All of DrScheme's tools are now available to the developer of a CGI script. For example, DrScheme's error reporting works properly. Suppose the developer forgets to deal with illegal inputs explicitly and instead relies on Scheme's primitives to read the submitted strings (all Web inputs are strings) as numbers. Then the program raises an exception for ill-formed inputs, and DrScheme highlights the place where the program raised the exception as if the program were an ordinary interactive program. See figure 4.8 for an illustration.

Consider the more complex example of DrScheme's single-step debugger [26]. The tool reduces Scheme programs according to Scheme's reduction semantics [41]. A developer may wish to use the stepper to understand the actions on a step-by-step basis. The stepper already accounts for library calls as atomic function calls, so that it properly displays transitions of CGI

programs—including input and output steps. See figure 4.9 for an illustration of this capability.

In general, developing CGI programs in the style of chapter 3 permits the use of conventional software engineering methods for interactive programs and the use of systematically enriched programming environments. This reuse thus bring order to the fragmented world of CGI programming.

4.5 Implementation Status

The CGI compiler exists in prototype form, while the CGI Teachpack for DrScheme is used in teaching introductory programming courses. Both the CGI compiler and the CGI Teachpack for DrScheme exist in prototype form. The prototype CGI compiler operates on R⁴RS [28] programs with some minor restrictions. I have developed a number of examples in this context, plus one full-fledged application: the teacher enrollment dialog for the TeachScheme! project.

The marshaling primitives use the existing PLT Scheme printer from the `print-convert` library, which automatically takes care of sharing and cycles in the reading and writing of environments and cookies. The encoding could benefit from a type-specific compression step [67] to reduce network traffic and the amount of data that is stored in cookies. The CPS conversion of Danvy and Filinski avoids introducing administrative beta redexes [33, 97].

```

(def ne box-0 (box 0))
(def ne box-1 (box 0))
;; main
(begin (set-box! box-0 (prompt-read "Enter the first number to add: "))
      (set-box! box-1 (prompt-read "Enter the second number to add: "))
      (show (+ (unbox box-0) (unbox box-1))))

```

Figure 4.6: A Stateful Interactive Program

```

(def ne-struct closure (code env))
;; Closure = (make-closure Int Env)
;; Env = (listof Value)

;; apply-closure : Closure (listof Value) *  $\rightarrow$  Value
(def ne apply-closure ...) ; as in figure 4.3
(def ne closures (vector ...))

;; replaced:
(def ne (prompt-read-k s k)
      (produce-html s (closure-code k) (closure-env k)))

;; added:
;; produce-html : String String (listof Value)  $\rightarrow$  void
;; effect: to write a CGI HTTP header and HTML Web form
;; including a cookie containing the-boxes
(def ne (produce-html question mark free-values)
      ...(write-boxes-to-cookie the-boxes)...)

(def ne bindings (get-bindings))

;; the-boxes : (vectorof Value), the current store
(def ne the-boxes
      (if (empty-bindings? bindings)
          (initialize-the-boxes)
          (read-boxes-from-cookie)))

;; initialize-the-boxes :  $\rightarrow$  (vectorof Value)
;; create a new store plus a sequence number

;; read-boxes-from-cookie :  $\rightarrow$  (vectorof Value)
;; turn a cookie into a store, check sequence number using a lock file

;; write-boxes-to-cookie : (vectorof Value)  $\rightarrow$  void
;; turn a store into a cookie, increment sequence number using a lock file

;; main
(cond
  [(empty-bindings? bindings)
   (apply-closure (make-closure 0 empty) (box 0))]
  [else (apply-closure
          (make-closure
            (string  $\rightarrow$  number (extract-bindings/single 'continue-at bindings))
            (create-env-from-strings (extract-bindings/single 'env bindings))
            (extract-binding/single 'response bindings)))]])

```

Figure 4.7: Its CGI Version

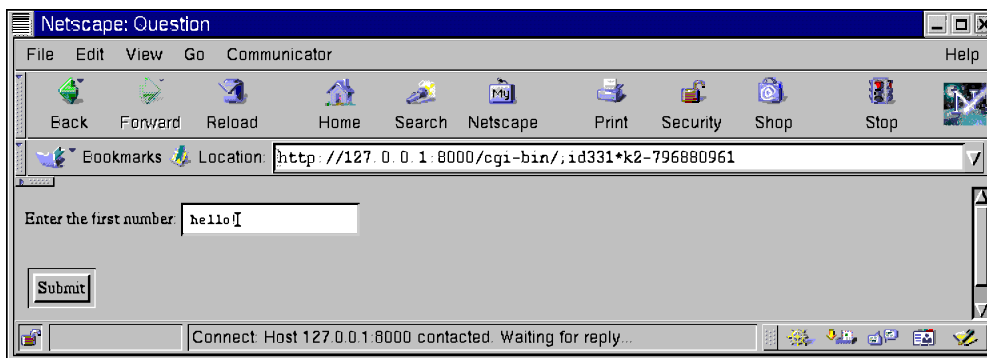
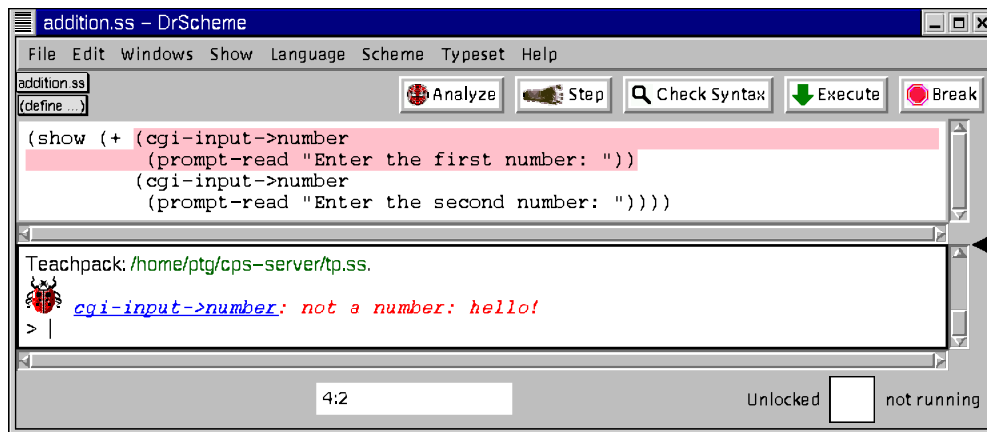
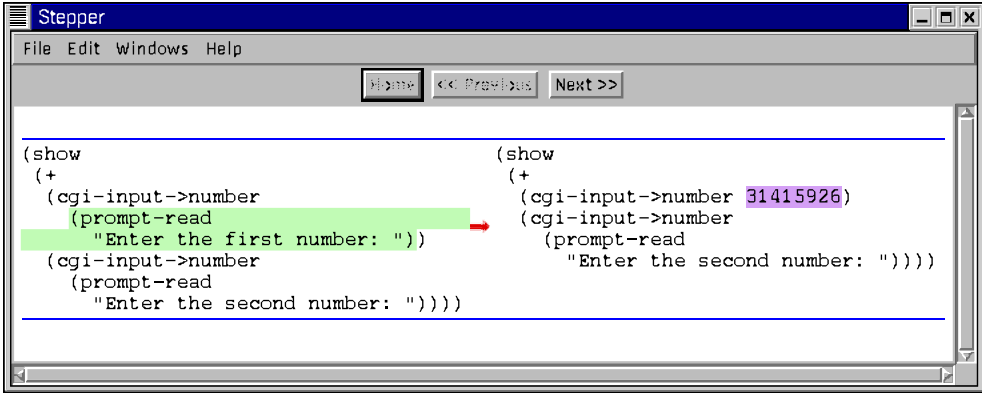


Figure 4.8: CGI Error Reporting



```
(show
(+
  (cgi-input->number
    (prompt-read
      "Enter the first number: "))
  (cgi-input->number
    (prompt-read
      "Enter the second number: "))))

(show
(+
  (cgi-input->number 31415926)
  (cgi-input->number
    (prompt-read
      "Enter the second number: "))))
```

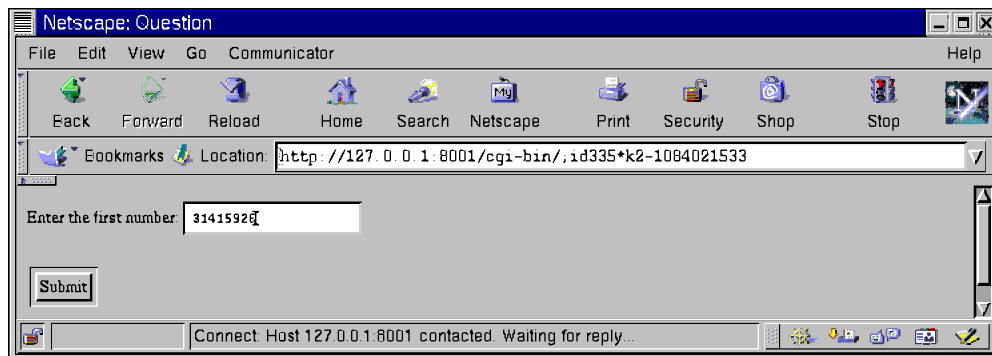


Figure 4.9: CGI Stepping

Chapter 5

Modeling the Web

5.1 Introduction

Chapters 3 and 4 raise the question of how the continuation based programming style compares to the more traditional script-centric approaches. The continuation style seems to eliminate errors in practice. What kinds of errors are unique to Web programming? Does an improved style eliminate these errors? Are there other solutions that address common problems in Web programming that apply to script-centric technologies as well? Answers to these questions appear in this chapter.

This chapter makes three contributions to the problem of designing reliable interactive Web programs. First, it develops a simple but formal model of Web interactions. This model can concisely explain the communication problem and the problem of Web forms becoming outdated. It also enables the study of solutions. Second, a type system solves the communication problem in a provable manner (relative to the model). Third, because not all the checks can be performed statically, a run-time check maintains store consistency.

5.2 Modeling the Web

Studying the problems of designing interactive Web programs benefits from a model with four characteristics. First, it consists of a single server and a single client in order to focus on the problems of simple sequential Web dialogs. Second, it deals exclusively with dynamically

W	$= S \times C$	$\{ \text{“”}, \text{“x”}, \text{“why”}, \text{“zee”} \}$	\subset	$String$
S	$= \Sigma \times P$	$\{ x, y, z \}$	\subset	Fid
P	$= Url \mapsto M^\circ$	$\{ www.drscheme.org, www.plt-scheme.org \}$	\subset	Url
M	$= programs$			
C	$= F \times \overrightarrow{F}$			
F	$= (\mathbf{form} \overrightarrow{Url} (\overrightarrow{Fid} \overrightarrow{V_b}))$			
V_b	$= Int \mid String$			

Figure 5.1: The Web

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 explained later, this suffices to represent the “Orbitz problem” and similar errors. Finally, to support experimenting with alternatives, the model is abstracted over the programming language. A λ calculus for forms and basic data serves as an example for this chapter.

The model lacks several properties orthogonal to the ones considered. First, the model ignores client-side storage, a.k.a. “cookies,” which primarily addresses customization and storage optimizations as discussed in section 4.3.2. Server-sided storage suffices for the purposes of this chapter. 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 [35, 49].

5.2.1 Server and Client

Figure 5.1 describes the components of the 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 5.2). 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.

The client consists of the current Web form and a set of all previously visited Web forms.

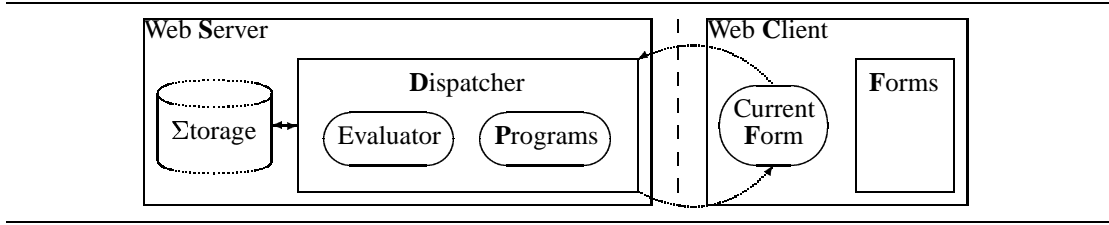


Figure 5.2: The Web Picture

$\text{fill-form} : W \rightarrow W$
 $\langle s, \langle (\mathbf{form} \ u \ \overline{(k \ v_0)}), \vec{f} \rangle \rangle \mapsto \langle s, \langle (\mathbf{form} \ u \ \overline{(k \ v_1)}), \{(\mathbf{form} \ u \ \overline{(k \ v_0)})\} \cup \vec{f} \rangle \rangle$

$\text{switch} : W \rightarrow W$
 $\langle s, \langle f_0, \vec{f} \rangle \rangle \mapsto \langle s, \langle f_1, \vec{f} \rangle \rangle$ **where** $f_1 \in \vec{f}$

$\text{submit} : W \rightarrow W$
 $\langle \langle \sigma_0, p \rangle, \langle f_0, \vec{f} \rangle \rangle \mapsto \langle \langle \sigma_1, p \rangle, \langle f_1, \{f_1\} \cup \vec{f} \rangle \rangle$
where $\langle \sigma_1, f_1 \rangle = d_p(\sigma_0, f_0)$

Figure 5.3: Transitions

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. 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 5.2 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 to the server. The server applies a 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 all visited forms.

Rewriting rules specify behavior by relating Web configurations. Figure 5.3 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 page¹ 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. Splitting the *submit* rule into two stages is reminiscent of structured operational semantics. [89] The actual dispatching and the evaluation are specific to the chosen programming language, which the next subsection introduces.

5.2.2 Functional Web Programming

Figure 5.4 specifies the WrForm programming language of Web Record-Forms. 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 5.1) in three ways: the syntax for forms, the syntax for terms (M), and the dispatch function d_p .

The **form** construct creates Web forms. The $M.Fid$ construct extracts the value of a form field with the name *Fid*. Two reductions [41], β_v and **select**, specify the semantics of WrForm.

The bottom half of figure 5.4 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.

5.2.3 Stateful Web Programming

Up to this point, scripts in the model can only communicate through forms. In practice, however, Web scripts often communicate not only via forms but also through external storage (files or

¹Returning to a non-cached page falls under the *submit* rule.

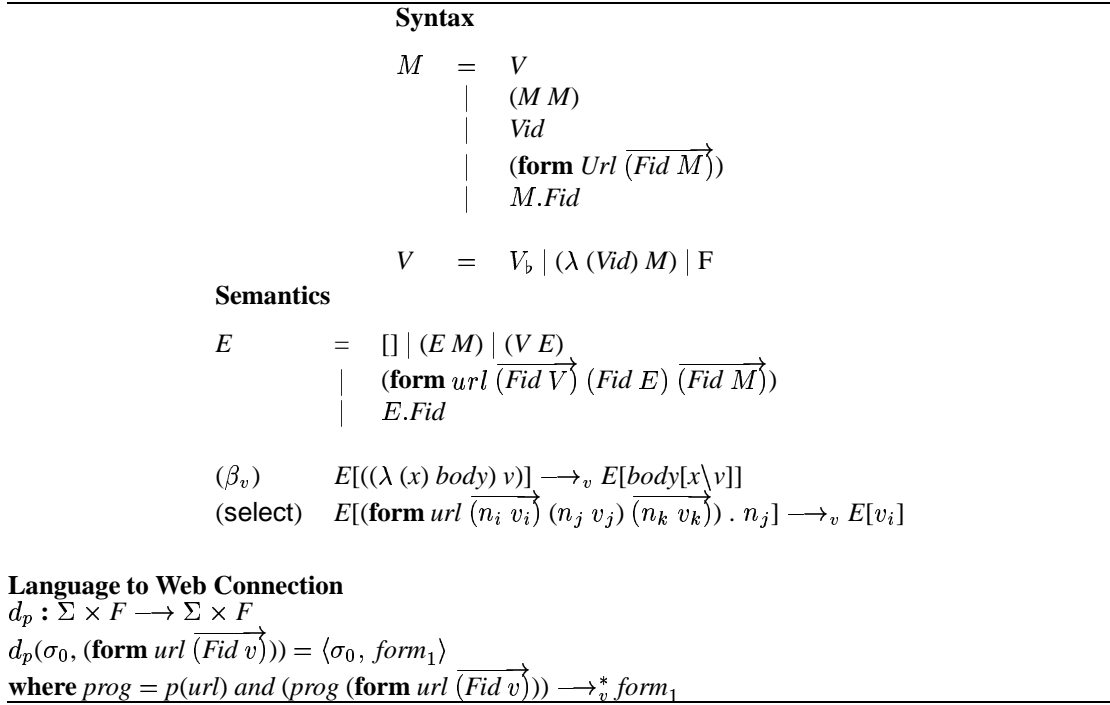


Figure 5.4: Web Programming Language

session objects). To model such stateful communications, **read** and **write** primitives extend WrForm. Figure 5.5 presents the language extensions. The two primitives empower programs to read flat values from and to write flat values to storage 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.

5.3 Problems with the Web

The model of Web interactions captures some common Web programming problems concisely. The first problem is that a Web script expects a different kind of form than is delivered. This is the “(script) communication problem.” The second problem reveals a weakness of the hypertext

Syntax	Language to Web Connection
$M = \dots \mid (\mathbf{read} \textit{Lid}) \mid (\mathbf{write} \textit{Lid} M)$	$\Sigma \sqsubseteq (\textit{Lid} \rightarrow V_v)$
<p>Semantics</p> $\frac{e_0 \rightarrow_v e_1}{\langle \sigma, e_0 \rangle \rightarrow_{v\sigma} \langle \sigma, e_1 \rangle}$ $\langle \sigma, E[(\mathbf{write} \textit{lid} v)] \rangle \rightarrow_{v\sigma} \langle \sigma[\textit{lid} \setminus v], E[v] \rangle$ $\langle \sigma, E[(\mathbf{read} \textit{lid})] \rangle \rightarrow_{v\sigma} \langle \sigma, E[\sigma(\textit{lid})] \rangle$ <p>where $\textit{Lid} \in \textit{dom}(\sigma)$</p>	$d_p(\sigma_0, (\mathbf{form} \textit{url} \overrightarrow{(\textit{fid} v)})) = \langle \sigma_1, \textit{form}_1 \rangle$ <p>where $\textit{prog} = p(\textit{url})$</p> $\langle \sigma_0, (\textit{prog} (\mathbf{form} \textit{url} \overrightarrow{(\textit{fid} v)})) \rangle \rightarrow_{v\sigma}^* \langle \sigma_1, \textit{form}_1 \rangle$

Figure 5.5: Language Extensions for Storage

transfer protocol. Due to the lack of an update method, information on client Web pages becomes obsolete and misleads the consumer. This is the “(HTTP) observer problem” indicating that HTTP does not permit a proper implementation of the Observer pattern [53].

5.3.1 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.² 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 5.6. 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

²A 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.

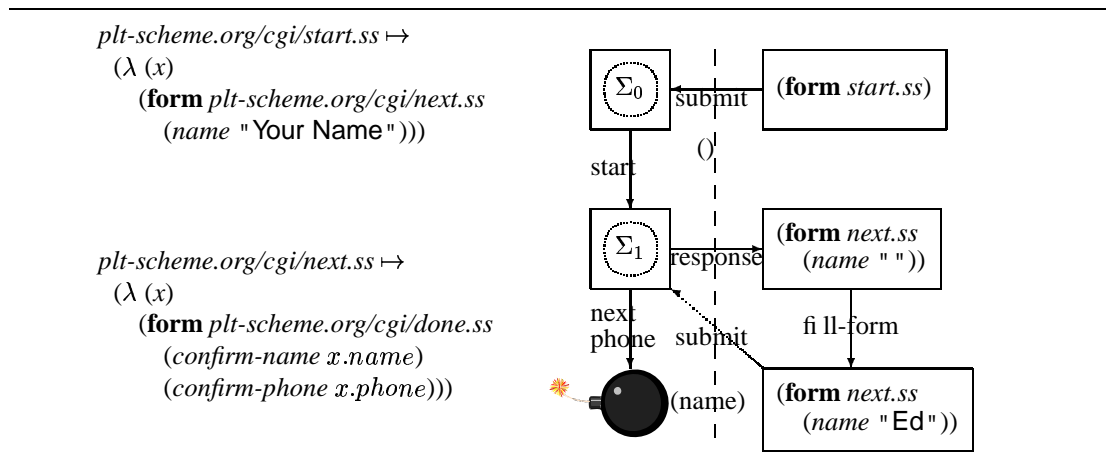


Figure 5.6: Collaborating Programs

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. The next section presents a type system that discovers such problems statically and still allows programmers to develop complex interactive Web programs in an incremental manner.

5.3.2 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

```

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 (generate-ack (read your-flight))))

```

Figure 5.7: Stateful Web Programs

the store when it really belongs in the Web form.

Figure 5.7 shows a reformulation of Orbitz’s problem (see section 1.1) 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 1.1. 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.

5.4 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). The Web model shows, however, that straightforward type checking doesn’t suffice, 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 on different Web servers and/or be written in different programming languages.

Typed WrForm therefore includes 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. This second step serves 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 deployment.

The type system for internal consistency checking heavily borrows from simply-typed λ -calculi with records [22, 86, 93]. Figure 5.8 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. The overloaded type environment maps both variables and store locations to types. An initial type environment Γ_0 maps locations in the external storage to flat types.³ Typed WrForm differs from WrForm only by requiring types for function arguments. That is, $(\lambda (x) M)$ becomes $(\lambda (x : \tau) M)$ 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 5.8, includes a set of constraints. A constraint $url : (\mathbf{form} \overrightarrow{(fid \ \tau_b)})$ insists that the program associated with *url* consumes Web forms of type $(\mathbf{form} \overrightarrow{(fid \ \tau_b)})$. The return type of the program associated with URLs do not affect the constraints because the Web programs are effectively, at least at the interaction points, in CPS.

Most type rules in figure 5.8 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

³The environment Γ_0 is fixed when beginning to check an individual program, but programmers may add extra locations for new programs.

Types	Type Judgments
$Type = Type \rightarrow Type$ $\quad \quad (\mathbf{form} \overrightarrow{(Fid Type_b)})$ $\quad \quad Type_b$ $Type_b = String \mid Int$	$\Gamma \vdash M : Type, \Xi$ <i>where</i> $\Xi = \{url : (\mathbf{form} \overrightarrow{(fid \tau)})\}$
Type Derivation Rules	
$\Gamma \vdash string : String, \{\}$ $\Gamma \vdash n : Int, \{\}$ $\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau, \{\}}$ $\frac{\Gamma, x : \tau_x \vdash m : \tau, \xi}{\Gamma \vdash (\lambda (x : \tau_x) m) : \tau_x \rightarrow \tau, \xi}$ $\frac{\Gamma \vdash m_0 : \tau_x \rightarrow \tau, \xi_0 \quad \Gamma \vdash m_1 : \tau_x, \xi_1}{\Gamma \vdash (m_0 m_1) : \tau, \xi_0 \cup \xi_1}$	$\frac{\Gamma \vdash m : (\mathbf{form} \overrightarrow{(fid_a \tau_{ba})}) (fid_x \tau_{bx}) \overrightarrow{(fid_b \tau_{bb})}, \xi}{\Gamma \vdash m.fid_x : \tau_{bx}, \xi}$ $\frac{\overrightarrow{\Gamma \vdash m : \tau_b, \xi_m}}{\Gamma \vdash (\mathbf{form} url \overrightarrow{(fid m)}) : (\mathbf{form} \overrightarrow{(fid \tau_b)}), \{url : (\mathbf{form} \overrightarrow{(fid \tau_b)})\} \cup \xi_m}$ $\frac{\Gamma(lid) = \tau_b}{\Gamma \vdash (\mathbf{read} lid) : \tau_b, \{\}}$ $\frac{\Gamma(lid) = \tau_b \quad \Gamma \vdash m : \tau_b, \xi}{\Gamma \vdash (\mathbf{write} lid m) : \tau_b, \xi}$

Figure 5.8: Internal Types for WrForm

expressions that generate constraints are **form** expressions.

The expression $(\mathbf{form} url \overrightarrow{(fid m)})$ 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 (possibly) 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 $url : (\mathbf{form} \overrightarrow{(fid \tau_b)})$, which must be checked later.

Figure 5.9 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.⁴ 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. Claim 1 asserts that individual Web scripts respond to appropriately typed requests without getting stuck.

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

Claim 1 is slightly unusual because the server type checks a program f , but when a request r arrives, the *submit* rule constructs a new term $(f\ r)$, which must be well typed. Thankfully, it is trivial to type check r because it is a value. It is also easy to check that if f has type $\tau_0 \longrightarrow \tau_1$ and r has type τ_0 , then $(f\ r)$ is well typed, with type τ_1 .

The proof would very likely follow the usual proofs of type soundness and strong normalization for the simply-typed λ -calculus. The proof of soundness typically consists of a type preservation lemma (reductions (\longrightarrow_v) preserve types) and a progress lemma (every well-typed term is either a value or reducible) [116]. See chapter 12 of Pierce's book [86] for details on proving strong normalization.

Extending claim 1 to the storage implementing $\longrightarrow_{v\sigma}^*$ relation does not break anything since the store only contains flat values. Even if an extension to WrForm broke strong normalization, this property is not critical. In this case, the other claims would weaken as usual to only apply when evaluation terminates. Investigating languages whose programs may execute indefinitely by handling arbitrarily many external events (such as Web requests) but must not become stuck in between events may be an interesting exercise, but this is an aside.

Claim 2 asserts that the server does not apply Web programs to forms of the wrong type, as long as the server starts in a good state. Stating the claim, though, requires an explanation

⁴Relaxing this restriction could allow forms to contain extra, unanticipated fields.

Server Extension and Additional Functions

$S = \Sigma \times P \times \Xi$ $\text{Install-program} : URL M W \longrightarrow W$ $\text{Install-program}(\langle \langle \sigma, p, \xi \rangle, c \rangle) = \langle \langle \sigma, p[url \setminus m], \text{Constrain}(\xi, url, m) \rangle, c \rangle$ when $\text{Consistent}(\text{Constrain}(\xi, url, m))$ $\text{Consistent} : \Xi \longrightarrow \text{boolean}$ $\text{Consistent}(\xi) \equiv$ $\begin{aligned} & (url : (\mathbf{form} \overrightarrow{(fid_0 \tau_0)})) \in \xi \wedge \\ & (url : (\mathbf{form} \overrightarrow{(fid_1 \tau_1)})) \in \xi \implies \\ & \overrightarrow{(fid_0 \tau_0)} = \overrightarrow{(fid_1 \tau_1)} \end{aligned}$	$\text{Constrain} : \Xi url M \longrightarrow \Xi$ $\text{Constrain}(\xi_0, url, m) =$ $\xi_0 \cup \xi_1 \cup \{url : (\mathbf{form} \overrightarrow{(fid_{in} \tau_{in})})\}$ where $\begin{aligned} \Gamma_0 \vdash m : (\mathbf{form} \overrightarrow{(fid_{in} \tau_{in})}) \\ \longrightarrow (\mathbf{form} \overrightarrow{(fid_{out} \tau_{out})}), \xi_1 \end{aligned}$
--	--

Figure 5.9: Constraint Checking

of 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:

$\text{server-typechecks}(\langle \sigma, p, \xi \rangle)$ iff $\text{Consistent}(\xi)$ and for each url in $\text{dom}(p)$,

$$\begin{aligned} \Gamma_0 \vdash p(url) : (\mathbf{form} \overrightarrow{(fid_1 \tau_{b1})}) \longrightarrow (\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})}), \xi_{url} \text{ and} \\ \xi_{url} \subset \xi \text{ and } url : (\mathbf{form} \overrightarrow{(fid_1 \tau_{b1})}) \in \xi \end{aligned}$$

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.

$\text{form-typechecks}(\langle \sigma, p, \xi \rangle, (\mathbf{form} \overrightarrow{url (fid v_b)}))$ iff

there are types $\overrightarrow{\tau_b}$ such that $\Gamma_0 \vdash v_b : \tau_b, \{\}$ and $url : (\mathbf{form} \overrightarrow{(fid \tau_b)})$ is in ξ and url is in $\text{dom}(p)$

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

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

To see that claim 2 holds, consider an arbitrary server $s_0 = \langle \sigma_0, p, z \rangle$ and an arbitrary form $f_0 = (\mathbf{form} \overrightarrow{url_0 (fid v)})$ such that $\text{server-typechecks}(s_0)$ and $\text{form-typechecks}(s_0, f_0)$. Since $\text{form-typechecks}(s_0, f_0)$, then url_0 must be in $\text{dom}(p)$. Let $prog = p(url_0)$. Again because the form typechecks, it is possible to choose types $\overrightarrow{\tau_b}$ such that $url_0 : (\mathbf{form} \overrightarrow{(fid \tau_b)})$. Since server-

$typechecks(s_0)$ and url_0 is in $dom(p)$ and $prog = p(url_0)$, it holds that

$$\Gamma_0 \vdash prog : (\mathbf{form} \overrightarrow{(fid_1 \tau_{b1})}) \longrightarrow (\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})}), \xi_{url_0}$$

Again since the server type checks, ξ must contain the constraint $url_0 : (\mathbf{form} \overrightarrow{(fid_1 \tau_{b1})})$.

From this, $(\mathbf{form} \overrightarrow{(fid_1 \tau_{b1})}) = (\mathbf{form} \overrightarrow{(fid \tau_b)})$ because ξ is consistent. The rule for type checking applications reveals that

$$\Gamma_0 \vdash (prog f_0) : (\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})})$$

By the stateful extension to claim 1, there are σ_1 and v_2 such that

$$\langle \sigma_0, (prog f_0) \rangle \longrightarrow_{v_\sigma}^* \langle \sigma_1, v_2 \rangle$$

Because of the preservation lemma typical of claim 1,

$$\Gamma_0 \vdash v_2 : (\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})})$$

The value v_2 must be a form f_1 since only form values have form types. Thus, $d_p(\sigma_0, f_0) = \langle \sigma_1, f_1 \rangle$. Hence, the submit reduction can happen without going wrong.

Claim 2 guarantees that one submission will work. It does not, however, guarantee anything about the next submission. An additional restraint ensures that the submission process preserves the preconditions of claim 2. Specifically, if the server's set of constraints is closed, the resulting configuration also guarantees the success of the next submission.

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

An extension of the same line of reasoning used for claim 2 applies to claim 3. Since s_0 and s_1 differ only in the values of their storage, it is easy to see that $server-typechecks(s_1)$. Choose url_2 such that $url_2 : (\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})})$ is in ξ_{url_0} . This constraint must exist because of the way type checking works in WrForm.

Specifically, type checking $prog$ produced an arrow type, so the derivation must have used the only rule that produces arrow types. The term the rule applied to must have been an abstraction, and the body of the abstraction has type $(\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})})$. Only two rules possibly produce form types. The rule for variable lookup can produce form types, but the variable must have been bound to a form somehow. Γ_0 does not contain any forms, so this is not a possibility. If the form returned by $prog$ is the same one that it received as its argument, then the submission clearly preserves *form-typechecks*. Otherwise, the form must be generated by $prog$ using a **form** expression type checked with the appropriate rule. The rule for forms produces the constraint $url_2 : (\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})})$ in ξ_{url_2} for some url_2 . To see that this constraint is in ξ_{url_0} requires a lemma that type checking collects all the constraints from subterms into the constraints for the entire term. This lemma would follow by induction of the structure of the type-derivation trees and a straight forward examination of each rule. Since $server-typechecks(s_0)$, all the constraints in ξ_{url_0} appear in ξ also. At this point, three items must be checked to see that $form-typechecks(f_1)$. First, the reasoning about claim 2 showed that the value f_1 has type $(\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})})$, so the values of the fields have types τ_{b2} . Second, $url_2 : (\mathbf{form} \overrightarrow{(fid_2 \tau_{b2})})$ is in ξ . Third, url_2 is in $dom(p)$ by the additional hypothesis of claim 3. Hence, $form-typechecks(s_1, f_1)$.

Alternative Web Programming Languages

It is not necessary to instantiate the model with a functional programming language. Instead, the model could have used a language such as <bigwig>, which is the canonical imperative while-loop language over a basic data type of Web documents [99]. Furthermore, the <bigwig> language already provides an internal type system that derives and checks information about Web documents. Its type system is stronger than WrForm's, allowing programmers to use complex mechanisms for composing Web documents.

The <bigwig> project and this chapter differ with respect to their ultimate goals. First, this chapter primarily accommodates the existing Web browser mechanisms. In contrast, <bigwig>'s

runtime system disables the back button, for example. Second, this chapter’s model aims to accommodate an open world, where scripts in ASP.NET, Perl, or Python can collaborate. The claims in this chapter 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 this constraint checking to dynamically typed languages requires a type inference system capable of determining the types of all possible forms a program might produce.

The model accommodates different programming languages without change. Using a language that provides record-style subtyping [113] for forms, however, would benefit programmers more if the subtyping affected the larger model as well. A few changes to the model could support record subtyping. First, the set of constraints on the server becomes two separate sets ξ_{formal} and ξ_{actual} . As the *server-typechecks* predicate shows, constraints arise from two situations. Type checking a program produces constraints on the *actual* form types passed to other programs with given URLs. Installing a program at a given URL associates the form type specified by the program’s *formal* parameter. Second, the *Consistent* relation insists that each URL appears only once in ξ_{formal} and that the set of form types in ξ_{actual} for that URL all be subtypes of the form type in ξ_{formal} . Third, the *form-typechecks* predicate requires that the received form be a subtype of the form type in ξ_{formal} . A slightly more restrictive version of *form-typechecks* would require the form type to be a member of ξ_{actual} , which would ensure that the form was generated by a known program.

5.5 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 [109]. Unfortunately, this solution doesn’t work in general for a number of reasons, some of which were discussed in the chapter on related work. Figure 5.10 shows a program written with WASH-CGI-1.0 (downloaded on October 8, 2002) that compiled without complaint using GHC-5.02.2 and “reserved” the wrong flight when run.

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. This approach is general, because it is independent of the scripting language. Dynamic checking is the appropriate compromise, because these kinds of situations depend very much 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 this model, 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**: $\Sigma \sqsubseteq Lid \rightarrow Time \times V_b$.

```

module Main where

import Prelude hiding (head)
import HTMLMonad
import Random
import CGI hiding (question )

import Persistent2 as P
import Maybe

main = run pickFlight

pickFlight =
  do Just initialHandle ← P.init "your-flight" " "
      ask (standardPage "Choose a Flight"
            (makeForm
              (do text "Departure Time: "
                  flight ← inputField (fieldVALUE 0)
                  submit (F1 flight)
                        (confirmFlight initialHandle)
                        (fieldVALUE " See Details" )))))

confirmFlight initialHandle (F1 flight) =
  do recentHandle ← P.current initialHandle
      Just nextHandle ← P.set recentHandle (value flight)
      ask (standardPage "Confirm Flight"
            (makeForm
              (do (text "Really reserve flight departing at ")
                  (text (value flight))
                  (submit F0 (receiptFlight nextHandle) (fieldVALUE "Confirm")))))

receiptFlight flightHandle F0 =
  do recentHandle ← P.current flightHandle
      flightTime ← P.get recentHandle
      htell (standardPage "Flight Receipt"
            (do (text "Reserved flight for ")
                (text flightTime)))

```

Figure 5.10: Outdated State in WASH/CGI

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

M	=	$\dots \mid (\mathbf{read} \textit{Lid} R) \mid (\mathbf{write} \textit{Lid} M R)$
R	=	relevant \mid irrelevant
$\{x, y, z\}$	\subset	\textit{Lid}

Figure 5.11: Relevance for Storage

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 .

This notion of whether a form is outdated or not is slightly strange. In particular, the system considers the form to be current when sent to the client even if some locations in the carrier set changed since the form’s creation. The timestamp checking catches only mismatches due to interactions between the execution of scripts, not within a single script.

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. The language therefore allows programs to specify whether reading or writing a location in the server state is a **relevant** or **irrelevant** action from the consumer’s perspective. This change enables the Web server to reduce the carrier set that it collects during a script execution and the number of warnings it issues. Figure 5.11 extends the model to address this issue by adding **relevant** and **irrelevant** annotations to **read** and **write** operations.

Extended with relevance annotations and types, the updated flight selection program in figure 5.12 reports an outdated form submission instead of booking the wrong flight. As seen in figure 5.14, the first execution of *confirm-flight.ss* produces a form containing $\sigma : 1$ and $\{ \textit{your-flight} \}$. The second execution of *confirm-flight.ss* produces a form with $\sigma : 2$ and $\{ \textit{your-flight} \}$. Switching back to the first form and submitting reveals that the server’s timestamp for *your-flight* is 2,

```

pick-flight.ss  $\mapsto$ 
  ( $\lambda$  ([empty-form : (form)])
    (form confirm-flight.ss
      (departure-time "nn:nn")))

confirm-flight.ss  $\mapsto$ 
  ( $\lambda$  ([first-form : (form (departure-time String))])
    (write your-flight first-form.departure-time relevant)
    (form receipt-flight.ss
      (confirm-time (read your-flight relevant))))

receipt-flight.ss  $\mapsto$ 
  ( $\lambda$  ([confirmed-form : (form (confirm-time String))])
    (buy-flight (read your-flight relevant))
    (form receipt-flight.ss
      (confirm-time (read your-flight relevant))))

```

Figure 5.12: Stateful Web Programs

```

 $\langle\langle\sigma_{-1}, p\rangle, \langle(\text{form pick-flight.ss (store\# -1) (carrier "[]"), \vec{f}\rangle\rangle$ 
 $\xrightarrow{\text{submit } w_0} \xrightarrow{\text{fill-form}}$ 
 $\langle\langle\sigma_0, p\rangle, \langle(\text{form confirm-flight.ss (store\# 0) (carrier "[]") (departure-time "5:50"), \vec{f}_0}\rangle\rangle$ 
 $\xrightarrow{\text{submit}}$ 
 $\langle\langle\sigma_1, p\rangle, \langle(\text{form receipt-flight.ss (store\# 1) (carrier "[your-flight]") (confirm-time "5:50"), \vec{f}_1}\rangle\rangle$ 
 $\xrightarrow{\text{switch } w_1} \xrightarrow{\text{fill-form}}$ 
 $\langle\langle\sigma_1, p\rangle, \langle(\text{form confirm-flight.ss (store\# 0) (carrier "[]") (departure-time "9:30"), \vec{f}_2}\rangle\rangle$ 
 $\xrightarrow{\text{submit}}$ 
 $\langle\langle\sigma_2, p\rangle, \langle(\text{form receipt-flight.ss (store\# 2) (carrier "[your-flight]") (confirm-time "9:30"), \vec{f}_3}\rangle\rangle$ 
 $\xrightarrow{\text{switch}}$ 
 $\langle\langle\sigma_2, p\rangle, \langle(\text{form receipt-flight.ss (store\# 1) (carrier "[your-flight]") (confirm-time "5:50"), \vec{f}_4}\rangle\rangle$ 
 $\xrightarrow{\text{submit}}$ 
 $\langle\langle\sigma_2, p\rangle, \langle\text{error-form}, \vec{f}\rangle\rangle$ 

```

Figure 5.13: Reductions

while the form was produced using version 1 of the *your-flight* location. The server detects the problem and notifies the client. Figure 5.13 contains a complete execution trace in the model. The trace shows all the interactions, the hidden time stamps in the forms, and the final error detection and notification.

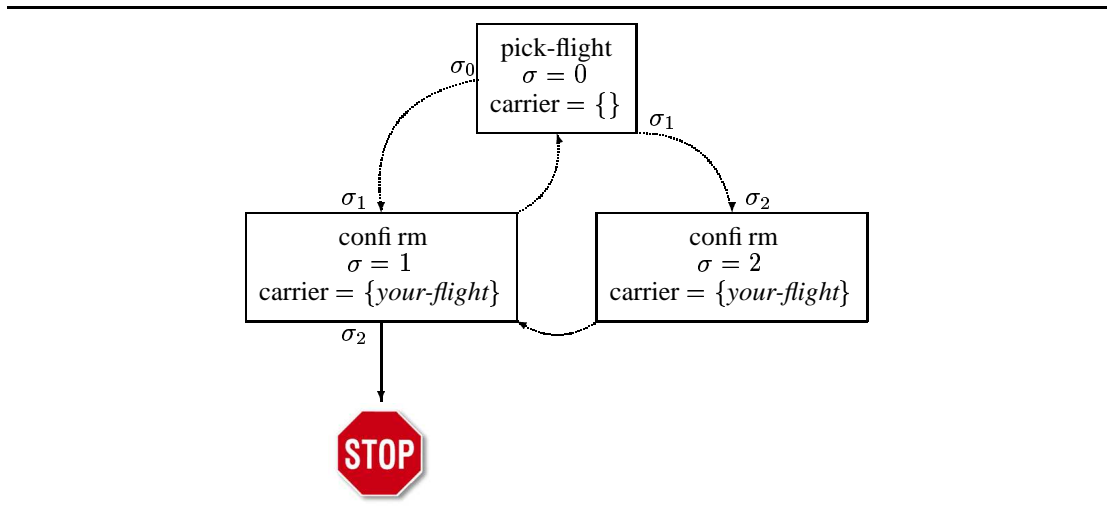


Figure 5.14: Avoiding Outdated Flights

Chapter 6

Beyond the Web

6.1 Introduction

Consider software that installs modern desktop applications such as Microsoft Office or Adobe Acrobat. The installer permits users to customize their installed components, choosing features and adjusting for the available disk space and other resource constraints. Specifically, these installers allow users to “browse” back and forth between options before they commit to a final configuration, in a manner reminiscent of Web browsers.

This simple interface raises several questions: What is the relationship between this browsing and that in a Web browser? Does the browsing power of these installers compare to that of Web browsers? If not, are there beneficial features that unify the two? Specifically, can consumers “bookmark” a GUI? And how do programmers methodically construct these programs? This chapter answers all these questions.

In the past few decades, interactive programs have greatly increased in complexity. Every decade appears to bring a whole new level of interface sophistication including full-screen ASCII-based graphics, pixel-based graphics, and the Web. While each improvement generally makes the quality of interaction better for the user, it also makes the programming task considerably more difficult. Much of the rigorous exposition on software design is still written for old-fashioned, inflexible textual interfaces [81].

Web software illustrates the problems that programmers must confront. Even though most

Web programs avoid the complexity of having to employ large and intricate graphical libraries, they must still contend with a myriad of user actions. In addition to linear traversals of Web sites, users commonly use Back buttons, clone windows, create and visit bookmarks, and so forth. Every one of these actions imposes new burdens on the developer.

Utilities such as wizards only minimally assist interactive software developers. Most of these programs largely deal with the tedious task of laying out and instantiating visual elements. They still leave the programmer to deal with the difficult problems of designing and composing the actual behaviors that lie beneath the “callbacks”. Unfortunately, most of the difficult tasks lie in the *interaction* between the visual elements and the underlying behavior. As a result, the wizards do not address many of the truly hard problems that arise in writing GUI programs.

The growing volume of interactive software makes these problems more urgent for software engineers and programmers. In particular, programmers need better tool support to assist in interactive software construction. As Myers says [81],

Tools influence the kinds of user interfaces that can be created. Successful tools use this to their advantage, leading implementers towards doing the right things, and away from doing the wrong things.

This chapter offers three contributions to the conceptual and development toolkits of GUI programmers. First, it provides a simple yet fruitful analysis of the fundamental flows of control (and data) in modern interactive programs. Applying this to Web and to GUI programs demonstrates that the former can often be much more complex than the latter. Specifically, cloning windows and bookmarking them endows users with very powerful interaction patterns lacking in GUIs. Second, a programming pattern helps developers provide these same features in GUIs, thereby enhancing the interaction facilities available to users. Third, it is possible to automatically transform programs written without these capabilities into ones that do provide them. A prototype implementation provided an opportunity to experiment with the new interaction

styles.

The rest of this chapter is organized as follows. Section 6.2.1 identifies similarities and differences in the browsing patterns engendered by Web and GUI applications. It also summarizes the features of the former that would be beneficial to the latter and outlines the difficulty in implementing the desirable GUI features. Section 6.3 presents the heart of this chapter: a programming pattern that overcomes this weakness. It then discusses the pattern's automation and correctness. Finally, it highlights additional sources of complexity this pattern handles. Section 6.4 describes two paths of information flow through programs and how the pattern accommodates both. The remaining sections discuss related work and conclude with directions for future work.

6.2 Usage Patterns in Interactive Programs

This chapter primarily addresses *large-grained interactive* software, meaning programs that

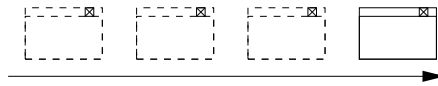
- present results to the user and seek additional inputs at several points during their execution;
- encourage a user- and event-driven model of computation rather than one where the agenda is dictated by the application;
- maintain a relatively large granularity of interaction.


A software installer, for instance, maintains an extremely large interaction granularity (ignoring individual character inputs, which libraries handle anyway); the granularity of a word-processor is extremely small (since its *raison d'être* is to handle character input). While, in theory, this chapter applies equally well to small-grained interactions, I have not applied them to such software, so it remains to be seen how they scale to that level.

6.2.1 Interaction Diagrams

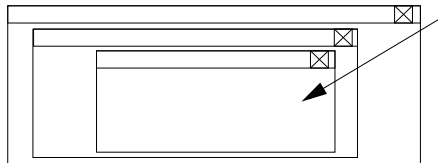
GUIs and Web-based programs are two of the most common and popular forms of interactive interfaces. To better understand interactive programs, this section first describes typical user interactions with GUIs, then contrasts these against Web interactions. Examining the “shapes” of control flow engendered by user interaction patterns sheds some light on the issue.


At its simplest, a series of modal GUI panes sequentially consume user information:

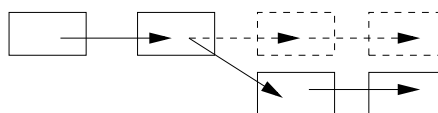


In this figure, each  represents a single GUI window. The arrow represents the passage of time. The dashed boxes represent windows that no longer exist on screen, because the user has already entered data and selected options. This perversely inflexible GUI program is little different from a rigid, old-fashioned textual program, where the software pre-determines a sequence of forms and the user has to provide the appropriate responses.

Most modern GUI programs, however, give their users the choice of closing a window to return to the previous one and make a different selection:



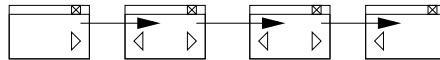
Here, time proceeds from the presentation of the largest window to the presentation of the smallest one; the 'es allow users to kill a modal window and alter their choices on a prior window. A more abstract diagrammatic representation captures this interaction. These *interaction diagrams* depicts the entire history of the interaction:



Each box stands for a window presented to the user. The dashed region indicates that the

user chose, then killed, one sequence of options, and has currently chosen a (possibly) different sequence, so the corresponding earlier windows are no longer visible on screen.

In some modern applications (including many installers), the GUI window includes explicit Forward and Back buttons. This gives the user the equivalent power of altering their decisions. While this has the benefit of offering a single window rather than crowding the screen with multiple ones, users need to use Back to see the options available and choices they made on previous versions of the window:



The resulting interaction diagram is the *same* as before (though earlier windows are effectively hidden beneath the Back button); the interface only provides a more convenient way of generating it, eliminating the need to pop up and kill numerous windows to explore different possibilities. In the end, while the user can explore a tree of choices, he can do this only one branch at a time.

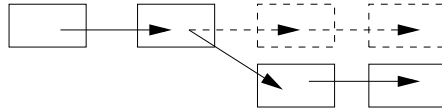
In contrast to GUI programming, Web programming, at its simplest, is a fairly straightforward task. CGI Web programs merely consume a set of name-value bindings and respond with a document of the appropriate type (which, currently, is often HTML). This description, however, proves to be excessively simplistic for almost all interesting Web applications, which have grown up from being “scripts” to programs.

Many of the user interactions with Web programs are similar to those with GUIs. Studying the shape of these interactions abstracts away details of libraries and protocols. At its simplest, the user simply clicks through a series of Web pages, resulting in this interaction diagram:



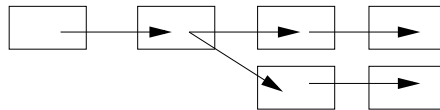
The difficulty arises when a user, unhappy with a set of choices, clicks the browser’s Back button to return to an earlier page and make a different choice. This effectively wipes out one thread of interactions and creates a new one. In this respect the browsing history is a stack, which is

represented explicitly by the Back and Forward buttons of many Web browsers, again capturing the fact that a user might have explored, but since abandoned, some sequence of options:



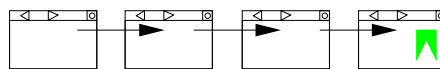
So far, Web programs seem to merely mimic the capabilities of GUIs. The Web, however, allows users to depart from, and enhance, these interaction styles in two important ways:

- First, users can clone the current window. The clones share a common past but, because they can now browse independently of one another, have different futures. Thus, cloning makes the tree structure of browsing explicit:

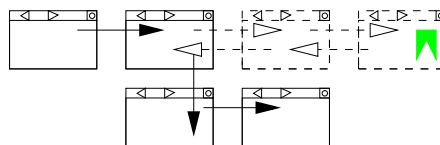


Importantly, each branch of the clone has solid lines and boxes, to indicate that both can run concurrently; this is *parallel exploration*.

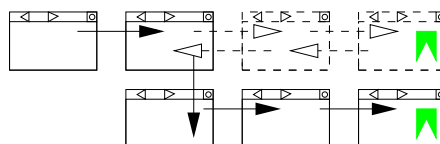
- Second, users can restore a prior exploration. That is, a user might initiate and bookmark a exploration as in this sequence of browser windows:



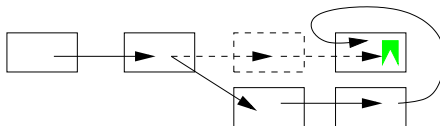
Then the user may explore an entirely different one,



but when invoking the bookmark restores the interface to its state at bookmark creation:



This results in the following interaction diagram:



Despite these interaction patterns, some people view Web applications as much simpler than GUI applications. This view seems to prevail for a few reasons:

- GUI programmers have to contend with staggeringly large interface toolkits [81]. Most Web applications generate little more than HTML using string processing.
- The relative youth of the Web means most applications have yet to reach the level of maturity of GUI programs. They perform fewer tasks, and thus don't deal with the same complexity of design demands.
- The complexity of GUI programs often ensues from the presence of explicit callbacks, concurrency and synchronization. These do not appear explicitly in Web programs, though failing to account for their implicit presence leads to erroneous interactions [91].
- Psychologically, users may be better conditioned to accepting erroneous behavior from Web applications, many of which run on remote sites and charge no direct fees, than from a GUI application that runs locally and charges the user a tangible up-front cost.

Web programs are, nevertheless, no simpler in *interaction flow* than GUI applications. For instance, every Web form submission is effectively a callback, with the CGI program or Java servlet referenced by the form's "action" field taking the place of a method invocation; cloning introduces concurrency. Furthermore, as the prior chapters of this dissertation illustrate, the Web already engenders *more complex* interaction flows than most GUI applications do, due to the features that browsers offer.

6.2.2 Desiderata

The complex interaction flows of the Web extract a price on the implementor, but translate into much greater convenience for the user. Users of GUI programs should benefit from these same

features. Consider the software installer mentioned earlier. Installers provide Forward and Back buttons for browsing the history of configuration choices. Large packages often include a “custom” installation mode, in which a user selects and de-selects individual components. When the user finishes choosing a configuration, the installer provides a summary of the space consumed for the components requested and confirms the configuration before it begins installation.

Suppose a user selects several components, but then finds the resulting installation would leave too little free disk space. As a result, he uses Back to return to an earlier menu, then navigates to a different configuration. With new choices made, perhaps the user wants to compare the disk usage to that consumed in the first configuration. Each time the user goes back and changes a configuration, however, he loses the previous one, as demonstrated by the GUI interaction diagrams. He therefore needs to have recorded the space usage for each prior configuration with some external utility.

Worse, suppose the user prefers an earlier configuration. He now must undo the currently active choices and somehow reproduce the preferred one. Therefore, he needs to have externally also recorded what components comprised each configuration. If the system includes several components, reproducing a configuration can be an extremely frustrating task.

This problem is not limited to installers. For example:

- Various “wizards” similarly guide users through application-specific choices. Many users find them frustrating because they limit user choices, but a flexible wizard need not do this: it can allow users to, at will, explore a large segment of the solution space, and to compare between solutions. Indeed, installers are merely simple forms of wizards.
- Many documentation browsers also don’t permit users to clone a browser or to bookmark it.
- Readers may have noticed similar limitations in point-of-sale software, both in stores and in dial-in services (such as airline reservations), which keeps sales agents from effectively

answering questions about multiple competing scenarios.

- This problem is even manifest in file browsing. Early versions of Windows opened a new filesystem browser each time a user entered a sub-directory. Because this can lead to a large number of open windows, later versions did not open a new window by default. A user can still open a new window on demand by Control+double-clicking on a folder, thus demonstrating the value of this interaction mode when made available in a controlled manner.

In short, this problem arises in numerous software contexts. The manual solution is time-consuming, error-prone, and ultimately distasteful because it forces the user to do what the computer ought to do instead. The problem is a mismatch between user need and software implementation. The goal of this chapter is to close the gap by bringing the implementation closer to the needs of the user.

In sum, GUI users should benefit from the same flexibilities that Web users currently enjoy. GUI users should be able to explore competing scenarios in parallel before committing to one (or more) of them. They should even be able to suspend the execution of a GUI by creating a “bookmark” and resume the computation whenever it is more appropriate or convenient.

6.2.3 Implementation Challenges

To understand the challenges a GUI implementor will face to provide these additional features, it is useful to review the problems Web programmers must confront. Web traversal patterns make strenuous demands on software developers, such as:

- The use of the Back button, which corresponds to backtracking, is a client-side event that does not notify the server (because the previous page is usually in the cache). This means the application’s view of the user’s location in the browsing tree—represented by the last page the user requested—may differ from the user’s actual location. Thus when the user submits a new request, the application may generate the wrong data, or may not even

understand the request. Fortunately, a GUI programmer can usually detect backtracking events, such as closing a window or clicking on Back in an installer.

- Users may initiate many more Web computations than they complete. A few users might “log out” or otherwise explicitly terminate the transaction, but many users simply abandon it. This forces developers to contend with large-scale, distributed resource management problems. In contrast, GUI users normally explicitly terminate unnecessary applications, at which point the operating system reclaims resources.
- Cloning the browser window creates the potential for concurrency, because the user can submit requests from both clones at virtually the same time. The browser does not inform the application about cloning, so a developer must always anticipate the possibility of race conditions. When GUIs allow users to perform parallel explorations, programmers will have to attend to the same synchronization needs.
- When users resume a computation by visiting a bookmark, they expect the application to remember the information they had provided at the time they created the bookmark. The developer must therefore determine how and where to store these data.

More of the problems of the Web will infiltrate GUIs as researchers begin to build *distributed* GUIs [51]. Meanwhile, problems such as the possibility of race conditions and having to remember data in a bookmark remain difficulties that a programmer building GUIs with flexible interfaces must surmount.

6.2.4 Stack Patterns in Flexible GUIs

Examining how representations of control information must evolve to accommodate additional functionality illustrates the challenges of implementing the complex control flows necessary for GUIs with flexible interfaces. The exposition uses an example that should be familiar to an academic audience: the paper submission portion of a conference manager. My co-advisor and I have developed and deployed a Web version of this program to manage several conferences.

The example here presents a greatly abridged form of its GUI counterpart. In each version of this example, the first window prompts for the author's personal information (name and email address). The second asks for a confirmation code (which is emailed to the address provided in the first window). The third window accepts information specific to the paper (title, abstract and filename). The last window prints an acknowledgment. The examples in this chapter use the Swing API [37] of Java [55], but the results are independent of the GUI library and the language.

```

public class SubmitPaper {
  public static void main(String[] args) {

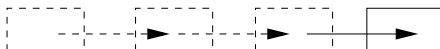
    System.out.println("Starting the Paper Submission Program. ");
    enterPaperInDatabase();
    System.out.println("Finished the Paper Submission Program."); }

  private static void enterPaperInDatabase() {
    String[] contactInfo = DirectGUI.promptRead("Paper Submission: ",
                                              new String[]{"Name: ", "Email: "});
    String name = contactInfo[0], email = contactInfo[1];
    String confCode = generateCode();
    Sntp.send(MAIL_SERVER, SMTP_PORT, FROM, email, buildMessage(name, confCode));
    String[] codeAttempt =
      DirectGUI.promptRead("Confirm Email",
                          new String[]{"Enter the confirmation code mailed to " + email});
    if (codeAttempt[0].equals(confCode)) {
      String[] paperInfo =
        DirectGUI.promptRead("Paper Info",
                            new String[]{"Title: ", "Abstract: ", "Paper Filename: "});
      /* update database */
      DirectGUI.show("Thank you for your submission."); }
    else { DirectGUI.error("Incorrect confirmation code"); }}
  private static String buildMessage(String name, String code) { /* ... */ }
  private static String generateCode() { /* ... */ }

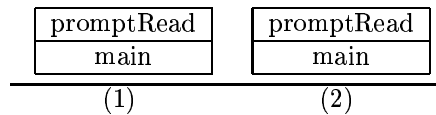
```

Figure 6.1: Inflexible Interface Version

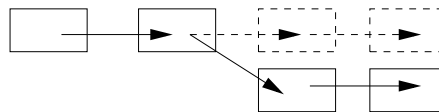
Studying programs that implement the interaction diagrams of section 6.2.1 provides a starting point for discovering how to increase these programs' flexibility. Figure 6.1 implements the first interaction diagram:



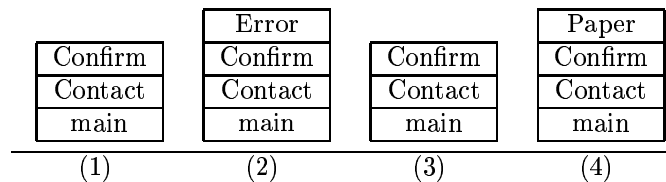
This program presents each of the submission windows in order. The program goes to lengths to avoid providing a convenient interface: each window simply returns the user's choice (by mutating a variable in its callback) instead of invoking a method that would generate the next input window. A given interaction with a user might result in the following stack snapshots (stacks grow upward), where each call to *promptRead* simply replaces the previous one:



Recognizing the weaknesses of this program, the developer chooses to implement the interaction diagram

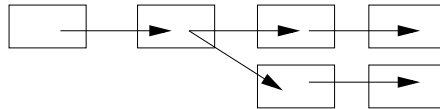


As detailed in section 6.2.2, this style of interaction is extremely common in a large number of applications ranging from software installers to filesystem browsers. The sample program, shown in figure 6.2, permits the user to close windows to return to prior windows and re-enter information. Examining the stack reveals the following:

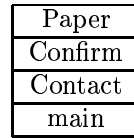


In the first snapshot, the user has entered the personal information, and is being prompted for the confirmation code. The second snapshot shows the error message frame that results from entering the wrong code. The user closes this window and returns to the previous one (3) to enter the correct code, which results in a paper submission dialog (4). This sequence demonstrates that the tree-shaped exploration maps to a stack by making sure only a linear thread of the exploration is active at any given time.

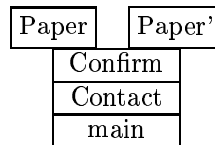
The next level of complexity in interaction diagrams is inspired by Web programs:



Suppose a programmer were able to implement this interaction mode. A user could begin to submit a paper

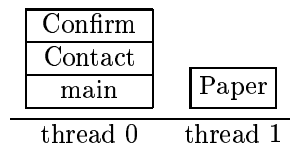


but then return to the window that spawns the paper information dialog and spawn a second such window, presumably to submit a second paper (the prime indicates that the frame refers to the same code, but may have different data):



This stack snapshot demonstrates the difficulty in implementing such a feature: the simple mapping from the interaction diagram to the stack breaks down, since it violates the linear nature of the stack.

The interaction diagram demands that the programmer generate multiple branches of concurrently active stack fragments. This naturally suggests threads, which offer multiple concurrent stacks. It is, however, difficult to solve this problem by using threads alone. Spawning a thread in the class *Confirm* to run *Paper* might generate the following stack configuration:



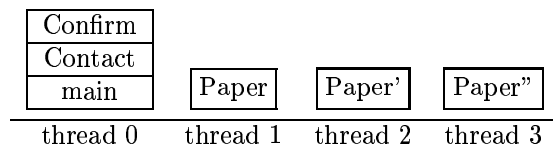
That is, threads generate entirely new stacks with no direct control flow to the spawning stack. In contrast, the interaction diagram demands stack *fragments* with a *common base*. The programmer thus becomes responsible for coordinating between the completion of the frame *Paper* and the resumption of *Confirm*. If the user wants to submit multiple papers, the stacks might resemble this:

```

public class SubmitPaper2 {
  public static void main(String[] args) {
    System.out.println(" Starting the Paper Submission Program. ");
    Lock lock = new Lock();
    enterPaperInDatabase(lock);
    synchronized (lock) { lock.wait(); }
    System.out.println(" Finished the Paper Submission Program. "); }
  public static void enterPaperInDatabase(final Lock lock) {
    GUI.promptRead(" Paper Submission: ",
      new String[]{" Name: ", " Email: " },
      new ProcessContact(lock)); }}
class ProcessContact extends Continuation {
  Lock lock;
  ProcessContact(Lock l) { lock = l; }
  void invoke(String[] contactInfo) {
    final String name = contactInfo[0], email = contactInfo[1];
    final String confCode = generateCode();
    Sntp.send(MAIL_SERVER, SMTP_PORT, FROM, email, buildMessage(name, confCode));
    GUI.promptRead(" Confirm Email ",
      new String[]{" Enter the confirmation code mailed to " + email},
      new ProcessConf(name, email, confCode, lock)); }
  private String buildMessage(String name, String code) { /* ... */ }
  private String generateCode() { /* ... */ }}
class ProcessConf extends Continuation {
  String name, email, confCode;
  Lock lock;
  ProcessConf(String n, String e, String c, Lock l) { name = n; email = e; confCode = c; lock = l; }
  void invoke(String[] codeAttempt) {
    if (codeAttempt[0].equals(confCode)) {
      GUI.promptRead(" Paper Info ", new String[]{" Title: ", " Abstract: ", " Paper Filename: " },
        new ProcessPaper(name, email, lock)); }
    else { GUI.error(" Incorrect confirmation code ");
      synchronized (lock) { lock.notify(); }}} }
class ProcessPaper extends Continuation {
  String name, email;
  Lock lock;
  ProcessPaper(String n, String e, Lock l) { name = n; email = e; lock = l; }
  void invoke(String[] paperInfo) { /* update database */
    GUI.show(" Thank you for your submission. ");
    synchronized (lock) { lock.notify(); }}} }

```

Figure 6.2: Version with Backtracking Support



To address this problem, a sentinel in frame *Confirm* checks a shared variable. When one of the spawned threads populates the shared variable, the sentinel resumes control in the primary stack, returning the value to the frame *Contact*. The following procedural pseudocode illustrates this idea:

```

processConfirm() {
    ... build window ...
    Lock l = new Lock ();
    JButton button = new JButton("Okay");
    button.addActionListener(new WakeOnClick(l));
    ... show window ...
    l.wait();
    extract fieldValues ...
    return fieldValues; }

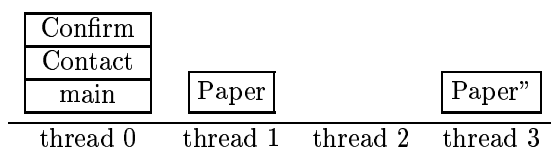
```

```

class WakeOnClick {
    Lock l;
    WakeOnClick(Lock lock) { l = lock; }
    void actionPerformed(ActionEvent e) {
        spawn-thread {
            processPaper()
            l.notify(); }}}

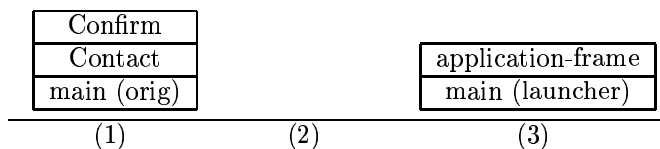
```

Besides its complexity, this solution has a *semantic* weakness. The sentinel returns when a thread (say the second one spawned) first writes to the shared variable, reducing the stacks to this configuration:



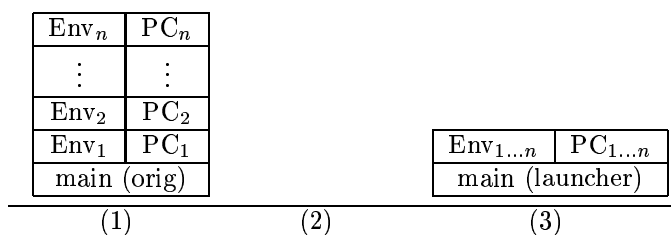
This means, however, that the sentinel is no longer present for the remaining threads. If the user decides to complete several of the spawned computations, only one of them can run to completion; the others languish for want of a sentinel. (While users will typically want to install only one configuration, section 6.3.1 explains why completing multiple scenarios is meaningful for many other applications.)

Inspired by the Web, the user might seek an even greater degree of convenience and power. Normally, aborting a program like a wizard or the paper submission suite forces users to restart the process from the beginning. Suppose, instead, a user enters the confirmation code, which leads to the paper information dialog, but realizes the paper is on a different filesystem. Rather than waste her labor, she may wish to bookmark the application itself, and quit its execution for now. To resume, she would use a special bookmark launcher. This generates the following stacks:



Snapshot (1) depicts the stack during bookmark creation. Snapshot (2) shows the empty stack, caused by exiting the application. Finally, snapshot (3) shows two frames. The bottom-most frame in a resumed program is always the bookmark launcher. The next frame is the first real frame of the resumed program.

This figure summarizes the responsibility that bookmarking computations thrusts on the programmer. They must ensure that the single resuming frame has enough information to perform the same computation that several frames did before termination. A more refined picture of the stacks in the general situation is



The Env_{*i*} refer to the lexical environments [7] of each frame, while the PC_{*i*} are the return addresses, representing the remainder of the computation. The single frame in snapshot (3) must conceptually have a lexical environment representing all the environments; likewise, its return address must conceptually refer to a computation that reflects all the actions that would have been performed by returning through the stack in snapshot (1). Furthermore, the user may resume the bookmark multiple times. It is simply not clear how to implement this feature using a combination of function calls and threads. Felleisen proves this task is not possible with local transformations [40]. The next section presents a unified approach that tackles all these problems.¹

¹Note that many applications permit the bookmarking of *data*. This is relatively easy: in the simplest case, it may involve remembering no more than a filename or some comparable string datum. This contrasts with bookmarking *the running application itself*. Section 6.3.1 returns to this issue.

6.3 Implementing Flexible GUIs: Control

Managing the complex control flows necessary for GUIs with flexible interfaces relies on a programming pattern that endows programs with these capabilities. This section also discusses its soundness and automation. It then touches on the relationship between control flows internal to the program and those imposed by the user.

6.3.1 Transforming GUI Programs for Flexibility

The previous section demonstrates that increased complexity of flexible interaction flows results in a corresponding complexity of *control flows*. Programmers need a methodical discipline to systematically convert a program of the form in figure 6.1 to one that supports parallel exploration and bookmarking. This section presents a transformation that meets this need.

The Transformation

As demonstrated above, in the extreme case, the stack at the resumption point needs to effectively package all the stack frames that were active at the point of bookmark creation so that the launcher can correctly resume the computation. The programmer thus requires:

- a new source program such that the stack's return address on resumption refers to code that indeed completes the computation remaining when the user created the bookmark;
- data structures that preserve information, such as the values of lexical variables, current at the time of bookmark creation.

Object-oriented languages make it especially convenient to bundle data with program fragments that operate on the data. This naturally suggests creating a class of **Continuation** [45, 103] objects that encapsulate the necessary data structures with the modified source to restore and continue the computation. A **Continuation** object provides a single method, *invoke*, which resumes the computation. Using **Continuation** objects, the programmer can methodically *generate* the bookmarkable GUI program from a non-bookmarkable version:

1. He creates a concrete sub-class of the **Continuation** class for every call site in the program (except invocations of most system library methods, which remain unchanged). The **Continuation** class for a call site has a field for each lexical value live in the method body after the call returns.
2. The code in the class's *invoke* method contains all the code in the method body beginning from when the call returns until the next transformed method invocation.
3. Method invocations gain an addition argument containing the code to continue executing. Method returns become calls to **Continuations'** *invoke* methods.

The only exception to this rule is at a potential bookmarking point. Here, the program does not call *invoke* in the **Continuation** object, since this would continue the computation immediately. Instead, the programmer supplies the **Continuation** object to the callbacks for the various buttons. Most of them run *invoke* when the user selects the corresponding button. A bookmark button's callback, alone, marshals the **Continuation** object instead.

Figure 6.3 presents the transformed version of figure 6.1. The transformation applies the steps above, and also changes the interface of *promptRead* to accept a **Continuation** object. This latter change permits *promptRead* to spawn subsequent windows. When a user cancels one of these windows, *promptRead* distinguishes this from the submission of information, and permits the user to make another selection. This permits the user to backtrack through the windows.

Close inspection reveals that the natural backtracking-friendly GUI code of figure 6.2 applies essentially the same transformation, partially and manually. The primary syntactic difference is that figure 6.3 uses inner classes to inline the class declarations. As the subsequent sections argue and demonstrate, the fully transformed program supports both parallel exploration and bookmarking.

Correctness and Automation

At this point, the presentation of the transformation has two significant weaknesses.

```

public class SubmitPaper3 {
    public static void main(String[] args) {
        System.out.println(" Starting the Paper Submission Program. ");
        enterPaperInDatabase(new Continuation_void() {
            void invoke() { System.out.println(" Finished the Paper Submission Program. "); } }); }
    public static void enterPaperInDatabase(final Continuation_void k) {
        GUI.promptRead(" Paper Submission: ", new String[]{" Name: ", " Email: " },
            new Continuation() {
                void invoke(String[] contactInfo) {
                    final String name = contactInfo[0], email = contactInfo[1];
                    final String confCode = generateCode();
                    Smtplib.send(MAIL_SERVER, SMTP_PORT, FROM, email, buildMessage(name, confCode));
                    GUI.promptRead(" Confirm Email ",
                        new String[]{" Enter the confirmation code mailed to " + email },
                            new Continuation() {
                                void invoke(String[] codeAttempt) {
                                    if (codeAttempt[0].equals(confCode)) {
                                        GUI.promptRead(" Paper Info ",
                                            new String[]{" Title: ", " Abstract: ", " Paper Filename: " },
                                                new Continuation() {
                                                    void invoke(String[] paperInfo) { /* update database */
                                                        GUI.show(" Thank you for your submission. ");
                                                        k.invoke(); } }); }
                                    else { GUI.error(" Incorrect confirmation code ");
                                        k.invoke(); } }); }
                                }
                            }
                        }
                    }
                }
            }
        }
    private String buildMessage(String name, String code) { /* ... */ }
    private String generateCode() { /* ... */ } }

```

Figure 6.3: Transformed Version

1. It has not offered any informal or formal reasoning of the two levels of correctness a programmer and user would care about:

Rudimentary Correctness That the transformed programs preserve the semantics of the original if the user never spawns parallel requests or creates a bookmark.

Extended Correctness That resuming a bookmarked program *does* correctly restore the state of the computation.

2. It has not discussed automation. The transformation involves a certain degree of manual labor. While it is methodical, it is tedious to perform manually, and the similarity between the original and the transformed version is clearly difficult to follow. This complicates both maintenance and debugging.

The following addresses these concerns in reverse order.

The transformation is clearly mechanical, and can therefore be implemented by a program rather than by a human. The prototype implementation for a subset of Java handles language features such as methods, conditionals and loops. As a result, programmers can develop and maintain the version in figure 6.1; the transformer generates code equivalent to (but not as readable as) figure 6.3.

Automating the transformation addresses many important problems that arise when applying programming patterns. Avoiding direct manipulation of the transformed program simplifies software maintenance. It does not, however, immediately eliminate concerns about debugging. After all, many of the bugs in GUIs arise precisely from complex control flows and interactions.

These transformations, however, do not introduce *new* errors into the program. The transformation has a strong theoretical foundation that could assist with proving that it preserves the program's semantics. Extending the transformer, would require a corresponding expansion of this reasoning. As a result, if the programmer is able to validate the behavior of the program with at most linear explorations, the transformer can extend this guarantee to parallel explorations and bookmarking.

This still leaves open the concerns about the fundamental correctness of the transformation itself. Thankfully, the literature already assures this: a careful reader will note that the transformation produces code essentially in *continuation-passing style*, which preserves a program's meaning. Continuations, combined with the store, capture the state of the computation. Our bookmarks save the continuation through the **Continuation** object, and the store by serializing it. This completely saves and restores the computation.²

²This intentionally ignores state such as open network connections. See the work by Zandy and Miller [117] for details on doing this.

Growing GUIs on Trees

This machinery suffices to now attack the problem of permitting parallel GUI explorations. The transformational approach based on CPS handles this problem naturally. By providing a data structure representation of the stack, the transformation permits the use of a tree-shaped “stack” without interference from the underlying architecture. The event queue of the GUI automatically queues multiple submissions.

Besides the benefits of simplicity and formality, a programmer has another important reason to prefer the transformation approach to the thread-based protocol of section 6.2.4. As described earlier, the thread-based implementation most easily permits the user to continue only *one* interaction branch past a function return—an implementation detail invisible and possibly counterintuitive to an unsuspecting user. Subsequent selections will go unheeded because the sentinel has no spawning stack left to resume.

For some applications, such as installers, permitting only one final selection is usually the correct semantics. In this case, the transformer can easily generate the appropriate code to implement *one-shot* semantics [63]. For many other applications, however, users should be free to take multiple choices through to completion. For instance, perhaps the user wants to book several related flights (the cities may be the same, but the dates may differ), or to generate multiple, related network configurations using the wizard (the email address and personal information are the same, but the choice of Simple Mail Transfer Protocol (SMTP) [90] server changes depending on the location).

Figure 6.4 demonstrates the use of multiple submissions in the running example. The order of forms in the conference submission program is deliberately chosen so an author needs to provide her contact information only once, and can then submit multiple papers. In the picture, she initiates the submission process by providing her contact information. She receives a confirmation code by email, which she enters into the validation window. She can now click on

the Validate button as many times as necessary. Each click offers the opportunity to submit a (different) paper. In this instance, she chooses to take both submission windows to completion, resulting in two conference submissions. The transformed program shares common prefixes of the stack, but invocation essentially duplicates these shared frames on demand.

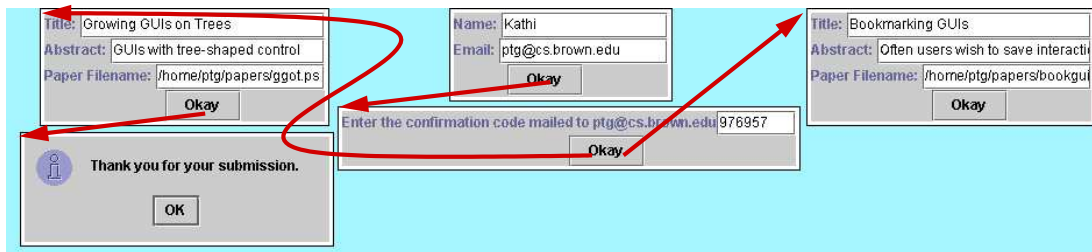


Figure 6.4: Multiple Submissions in Action

Bookmarking GUIs

The transformation of section 6.3.1 handles bookmarking also, and indeed is inspired by the need to do this. In particular, the transformer can automatically generate the callback for a Bookmark button. Clicking Bookmark prompts the user for a filename, to which the application writes the serialized [104] continuation object. For example, consider this modified fragment of *promptRead* used in figure 6.3. The transformer replaces the bookmark button's callback with a *Serializable* instance of Swing's *ActionListener* [37], where *frame* is the frame that contains the continuation object:

```
new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String fileName = /* pick bookmark file name */ ;
        FileOutputStream baos = new FileOutputStream(fileName) ;
        ObjectOutputStream oos = new ObjectOutputStream(baos) ;
        oos.writeObject(frame) ; }}}
```

The corresponding bookmark launcher is

```
public class LoadBookmark {
    public static void main(String[] args) {
        FileInputStream fin = new FileInputStream(args[0]) ;
        ObjectInputStream ois = new ObjectInputStream(fin) ;
        JFrame frame = (JFrame)ois.readObject() ;
```

```
frame.show() ; }}
```

Figure 6.5 shows screen-shots of this behavior. In the left panel, the user begins a paper submission and bookmarks the application (in this case, the bookmark file is roughly 15Kb in size). In the right panel, the user resumes the bookmark on the appropriate machine to finish submitting the paper.

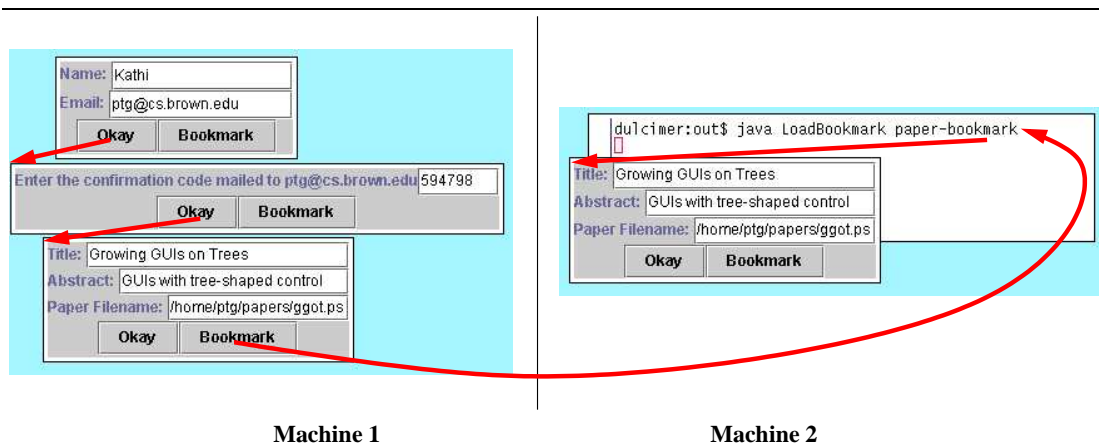


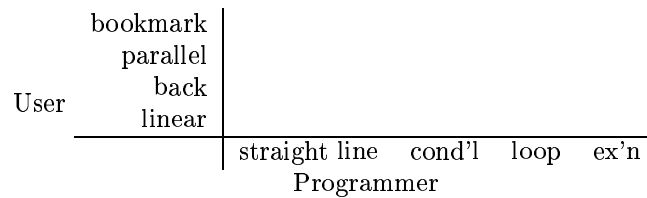
Figure 6.5: Bookmarking in Action

6.3.2 Two Dimensions of Complexity

The exposition above deals with the conversion of conventional GUI programs into more flexible variants. The program figure 6.1 presents is, however, rather simplistic. The actual conference software has a loop, to check for the syntactic correctness of the email address, and catches exceptions raised by trying to email erroneous addresses. Although the presentation elides these details, but these and other features would be found in any realistic GUI program.

The simplicity of the source program compared to the corresponding complexity of the control flows discussed in section 6.2.4, demonstrates that there are *two* sources of complexity in the interaction flow of flexible GUI programs. The internal dimension is controlled by the programmer, and usually handles the processing of data (for instance, a loop that iterates over elements in a collection). The external dimension is generated by the user's requests, and hence

lies outside the control of the programmer:



As the complexity grows in the internal dimension also, it becomes extremely difficult to satisfy the demands of both dimensions. This makes a methodical means of program construction that accounts for the various kinds of external control flows especially valuable to programmers. The techniques of this chapter offer programmers this support.

6.4 Implementing Flexible GUIs: Data

This chapter has thusfar discussed the subtleties of control flow in programs. Programs also have different patterns of *information flow*. Broadly, there are two kinds of information, which are familiar to authors of interpreters and compilers: *lexically updated* and *globally mutated* information, which reside, respectively, in the *lexical environment* and in the *mutable store* [7, 50]. Automatically generating code for parallel exploration must distinguish between these so there are multiple copies of the former, but only one of the latter.

In a functional programming language such as ML, it is usually easy to distinguish between these two: by default information resides in the environment, unless it has type **ref**. Only variables of type **ref** are subject to mutation. In a language like Java, however, all variables receive values through mutation, even during initialization. This uniformity of syntax hides the fact that some variables are mutated only once, because of their lexical nature. Examining which variables have static single assignment [31] identify this case; these are the lexical ones. This analysis is much simpler for Java than for languages supporting call-by-reference due to the lack of variable aliasing.

The transformer is sensitive to this distinction between the environment and the store. Lexically updated variables reside in **Continuation** objects, which close over their values. There are

hence as many copies of these variables (represented as **Continuation** object fields) as there are parallel explorations. In contrast, the transformation lifts mutated variables so that all continuations at a call site share the same ones. Generating bookmarks must correspondingly maintain this distinction.

In a certain sense, all non-trivial GUI programs offer a weak version of bookmarking, in the form of configuration files. This common feature has two major shortcomings:

1. The configuration file is global, and thus shared by all executions of the program. (For instance, changing the default identity of the author in a word-processor affects all subsequent documents.) To work around this, a few programs permit multiple configurations (“profiles”).
2. Programs sometimes incorrectly restore configuration data. They usually avoid this problem by not capturing complex configuration actions or ones that are not easy to marshal into files. This often forces users to manually recreate these configurations every time they use the program.

The second problem is automatically addressed by the transformations, because bookmarks reflect the complete state of the program. The first problem results from an incomplete understanding of the distinction between lexical and global information flows; the transformations automatically account for this distinction, saving the programmer from having to explicitly create support for profiles. Each bookmark thus behaves like a profile by keeping its lexical information separate from other bookmarks. For example, an academic reviewer could create a customized copy of a word-processor that hides her identity, which she can then use to annotate submissions with anonymous feedback.

Chapter 7

Conclusions

Constructing programs that dialogue with the consumer in a direct style produces more uniform, robust results for the consumer with less effort from the programmer. This style avoids error-prone manual marshaling and unmarshaling of data by programmers in between interactions with the consumer. The programs are easier to understand and maintain.

7.1 Web Dialogues via a Server

One way to support this direct-style programming paradigm involves a custom Web server. By capturing the current continuation each time the program interacts with the consumer, the server can associate the control state of the servlet with a URL. Requests for these URLs return the servlet to the context of the interaction that created them.

Storing continuations for all the servlets inside the Web server demands tight integration between the servlets and the server. The server dynamically loads multiple servlets into the same address space and provides them with services in the form of Web interaction primitives. Tight integration with each other enables servlets to share values easily through shared lexical scope. Tight integration with the server keeps the exchange of requests and responses efficient, avoiding process creation or interprocess communication overhead.

Supporting tight integration by rejecting the underlying operating system's process abstraction comes at a cost. The server must manage the servlets' resources instead, in effect becoming

its own operating system. To this end, the server exploits several of MrEd's primitives for managing resources: threads, custodians, parameters, and units or namespaces. These primitives enable the server to maintain close sharing of data efficiently while reclaiming resources.

The server also provides servlets with primitives *send/forward* and *send/finish* to reclaim resources manually. These mechanisms not only recycle resources sooner, but they also prevent problems such as double billing.

The server works well enough in practice to serve several Web sites, including the site for DrScheme [44], a couple of online texts [1, 3], the TeachScheme! project [2], and several programming languages related workshops [70].

7.2 Web Dialogues via Compilation

Despite the advantages the Web server offers, it may not be ideal for every situation. Storing continuations on the server in between interactions may consume too much server-side storage for some widely used applications. Also, some Web administrators may already be committed to another server. Keeping all the continuations in one process also complicates distributing the Web server across multiple machines [73].

In such situations, developers need to use an existing server with a standard interface for Web programming and keep state needed for clients' sessions on the clients' machines. To meet these needs, my dissertation also shows how techniques for compiling functional programming languages provide the foundation for transforming interactive Web programs into standard CGI scripts. These scripts run on any CGI-compliant Web server and store all the necessary state in the browsers' Web pages or cookies. The techniques can be used for other languages by either writing a compiler or by carefully performing the transformations by hand in a principled manner.

The compilation-based implementation suffers several drawbacks. It lacks efficiency due to the underlying CGI protocol. Servlets may not communicate with each other easily, especially

when sharing closures, semaphores, or other difficult to marshal values. It also introduces several security concerns, although well-known cryptographic techniques address these concerns.

Since every little language or language extension needs a development environment, an extension to DrScheme enables programmers to re-use all of its development tools for Web programming.

7.3 Model of Web Dialogues

This dissertation also develops a foundational model of Web interactions. Although the helpfulness of *send/suspend* motivated the formal investigation, the model focuses on standard Web interactions which do not exploit the power of continuations. Each Web program consists of many individual scripts which communicate through Web forms and external storage.

The model reveals two kinds of problems that arise in Web programming. Individual scripts may miscommunicate through forms, and they may process outdated information from old Web forms. A type system solves the first problem, while a dynamic check addresses the second.

Even if each individual Web script could be correct in the appropriate context, the connections between scripts through Web forms may fail. An extended type checker tracks both the expected forms consumed by scripts and the possible forms produced (and sent to other scripts). With this information, the server can eliminate these inter-script communication errors. Although in practice the continuation based programming paradigm eliminates much of the need for communication through forms, it does not forbid such errors entirely.

Preventing servlets from processing outdated forms requires a dynamic check. Attaching a timestamp and a carrier set to each page and a timestamp to each store location enables the server to detect outdated submissions. If annotations on read and write operations indicate whether or not the store access is relevant or irrelevant.

7.4 GUI Dialogues

The chapter on GUIs is motivated by the simple desire to compare the browsing power of different kinds of interactive software. In particular, it compares the patterns available to Web site users with those provided by many interactive graphical programs, using software installers as a representative example of these applications. Abstracting away the details of Web and GUI toolkits reduces the use of these applications down to simple diagrammatic representations.

This analysis demonstrates that, despite their numerous interface benefits, GUIs fall short of Web interfaces by failing to support key interaction patterns. In particular, Web users can both clone windows and create bookmarks while browsing, both of which are rarely if ever supported by graphical applications. As illustrated, a developer must overcome several implementation burdens to provide this support. This support ensues automatically by transforming programs into an extension of continuation-passing style, a pattern more commonly found in the back-ends of compilers for advanced functional languages. The transformation of a sample application presents this pattern in action.

Chapter 6 opens numerous avenues for future research. The most obvious question is how well it scales to interactive programs such as word-processors, where the granularity of interaction is at the level of a single keystroke. Depending on the structure of the code, bookmarks could either be too sparse (recording no past history) or too dense (recording all past keystrokes) with information. This may necessitate more forgiving forms of the transformation.

Second, a static analysis could help keep the size of bookmarks modest. In principle, it should be possible to reduce the size of bookmarks of the style described in section 6.4 to be very close to that of manually-constructed configuration files. As an initial effort, the implementation can accomplish a similar end with much less sophistication by factoring out data common to multiple bookmarks, thus separating bookmarks into control and lexical data.

Ultimately, these transformational ideas are best incorporated into interface-generation tools

such as wizards. The user traversal patterns an interface generates should be kept separate from its layout, and the synchronization and other needs imposed by the traversal should be generated automatically by a tool. Chapter 6 presents a foundation for offering very general support of this form.

Continuations facilitate the construction of non-Web dialogues as well. Installation and configuration programs request information from the consumer through a series of dialogues. The sequences of interactions helpful to consumers mimic those demanded by the navigation features of Web browsers. All of the techniques for enabling advance navigation (parallel exploration, user-directed backtracking, and returning to a bookmarked location later or on a different machine) apply not only to Web programming, but also to GUI dialogues found in other programs.

Well-known compilation techniques can transform a program that uses a dialogue-based GUI library to support Web-like interactions. The bookmark files that record the program's progress need not be larger than the corresponding files implemented in an ad hoc manner by some installation programs.

Bibliography

- [1] <http://www.htdp.org/>.
- [2] <http://www.teach-scheme.org/>.
- [3] <http://www.htus.org/>.
- [4] Keyed-hash message authentication code (HMAC). <http://www.nist.gov/hmac>.
- [5] Acme Labs. Web server comparisons.
<http://www.acme.com/software/tthttpd/benchmarks.html>.
- [6] Adya, A., J. Howell, M. Theimer, B. Bolosky and J. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference*, 2002.
- [7] Aho, A. V., R. Sethi and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., Reading, Mass., 1986.
- [8] Apache. <http://www.apache.org/>.
- [9] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.
- [10] Arlitt, M. and C. Williamson. Web server workload characterization: the search for invariants. In *ACM SIGMETRICS*, 1996.
- [11] Aron, M., D. Sanders, P. Druschel and W. Zwaenepoel. Scalable content-aware request distribution in cluster-based network servers. In *Annual Usenix Technical Conference*, 2000. San Diego, CA.
- [12] 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.

- [13] Bakken, S. S., A. Aulbach, E. Schmid, J. Winstead, L. T. Wilson, R. Lerdorf, A. Zmievski and J. Ahto, January 2002. <http://www.php.net/manual/>.
- [14] Banga, G. and P. Druschel. Measuring the capacity of a Web server. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [15] Banga, G., P. Druschel and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Third Symposium on Operating System Design and Implementation*, February 1999.
- [16] Biagioni, E., K. Cline, P. Lee, C. Okasaki and C. Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, 1998.
- [17] Borland Software Corporation.
<http://www.borland.com/jbuilder/>.
- [18] Brabrand, C., A. Møller, A. Sandholm and M. I. Schwartzbach. A runtime system for interactive Web services. In *Journal of Computer Networks*, 1999.
- [19] Bray, T., J. Paoli and C. Sperberg-McQueen. Extensible markup language XML. Technical report, World Wide Web Consortium, February 1998. Version 1.0.
- [20] BrightPlanet. DeepWeb.
<http://www.completeplanet.com/Tutorials/DeepWeb/>.
- [21] Bryant, A. and J. Fitzell. Tutorial: A walk on the seaside.
<http://beta4.com/seaside2/tutorial.html/>.
- [22] Cardelli, L. Type systems. In *Handbook of Computer Science and Engineering*. CRC Press, 1996.
- [23] Cardelli, L. and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*. Springer-Verlag, Berlin Germany, 1998.
- [24] Carlsson, M. and T. Hallgren. Fudgets—a graphical user interface in a lazy functional language. In *Functional Programming and Computer Architecture*, 1993.

- [25] Chandra, S., B. Richards and J. R. Larus. Teapot: Language support for writing memory coherence protocols. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–248, May 1996.
- [26] Clements, J., M. Flatt and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, 2001.
- [27] Clinger, W. and J. Rees. Macros that work. In *ACM Symposium on Principles of Programming Languages*, pages 155–162, 1991.
- [28] Clinger, W. and J. Rees. Revised⁴ report on the algorithmic language Scheme. In *ACM Lisp Pointers*, pages 1–55, 1991.
- [29] Coward, D. Java servlet specification version 2.3, October 2000.
<http://java.sun.com/products/servlet/>.
- [30] Crocker, D. H. RFC 822: Standard for the format of ARPA internet text messages, August 1982.
<http://www.ietf.org/rfc/rfc822.txt>.
- [31] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
- [32] Daemen, J. and V. Rijmen. Advanced Encryption Standard.
<http://www.nist.gov/aes>.
- [33] Danvy, O. and A. Filinski. Representing control: a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.
- [34] Demsky, B. C. An empirical study of technologies to implement servers in Java. Master’s thesis, MIT, May 2001.
- [35] Dierks, T. and C. Allen. RFC 2246: The transport layer security protocol, January 1999.
<http://www.ietf.org/rfc/rfc2246.txt>.

- [36] Drepper, U. and I. Molnar. The native POSIX thread library for Linux, January 2003.
<http://people.redhat.com/drepper/nptl-design.pdf>.
- [37] Eckstein, R., M. Loy and D. Wood. *Java Swing*. O'Reilly, 1998.
- [38] Eich, B. EcmaScript language specification, December 1999. Standard ECMA-262.
- [39] Elliot, C. and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, 1997.
- [40] Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming*, 17(1–3):35–75, December 1991.
- [41] 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.
- [42] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1, June 1999.
<http://www.ietf.org/rfc/rfc2616.txt>.
- [43] Filliatre, J.-C. and D. de Rauglaudre. Objective Caml library for writing CGIs, 1998.
<http://www.lri.fr/~filliatr/ftp/ocaml/cgi/>.
- [44] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 2001.
- [45] Fischer, M. J. Lambda-calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109, Los Cruces, 1972.
- [46] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

- [47] Flatt, M. and M. Felleisen. Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
- [48] Flatt, M., R. B. Findler, S. Krishnamurthi and M. Felleisen. Programming languages as operating systems (*or*, Revenge of the Son of the Lisp Machine). In *ACM SIGPLAN International Conference on Functional Programming*, pages 138–147, September 1999.
- [49] 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>.
- [50] Friedman, D. P., M. Wand and C. T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, 1992.
- [51] Fuchs, M. *Dreme: for Life in the Net*. PhD thesis, New York University, 1996.
- [52] Fünfroeken, S. Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In *Proceedings of the Second International Workshop on Mobile Agents*, pages 26–37. Springer-Verlag, September 1998. LNCS 1477.
- [53] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [54] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [55] Gosling, J., B. Joy and G. L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.
- [56] Graham, P. Beating the averages. <http://www.paulgraham.com/avg.html>.
- [57] Graham, P. Lisp in web-based applications.
<http://www.paulgraham.com/articles.html>.
- [58] Graunke, P. Programming the Web with high-level programming languages. Master's thesis, Rice University, April 2001.

- [59] 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, November 2001.
- [60] Graunke, P., R. B. Findler, S. Krishnamurthi and M. Felleisen. Modeling Web interactions. In *European Symposium on Programming*, April 2003.
- [61] 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.
- [62] Graunke, P. T. and S. Krishnamurthi. Advanced control flows for flexible graphical user interfaces or, growing GUIs on trees or, bookmarking GUIs. In *ACM International Conference on Software Engineering*, pages 277–287, 2002.
- [63] Haynes, C. T. and D. P. Friedman. Constraining control. In *12th ACM Symposium on Principles of Programming Languages*, pages 245–254, 1985.
- [64] Hughes, J. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.
- [65] International Business Machines, Inc.
<http://www.ibm.com/websphere>.
- [66] Jackson, M. A. *Principles of Program Design*. Academic Press, 1975.
- [67] Jansson, P. and J. Jeuring. Polytypic compact printing and parsing. In Swierstra, S. D., editor, *European Symposium on Programming*, pages 273–287. Springer-Verlag, 1999.
- [68] Johnsson, T. Lambda lifting: transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985.
- [69] Kelsey, R., W. Clinger and J. Rees. Revised⁵ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

- [70] Krishnamurthi, S. The CONTINUE server (or, how I administered PADL 2002 and 2003). In *ACM Symposium on Practical Aspects of Declarative Languages*, pages 2–16, 2003.
- [71] Krishnamurthi, S., K. E. Gray and P. Graunke. Transformation-by-example for XML. In *ACM Workshop on Practical Aspects of Declarative Languages*, January 2000.
- [72] Kristol, D. and L. Montulli. RFC 2109: HTTP state management mechanism, February 1997.
<http://www.ietf.org/rfc/rfc2109.txt>.
- [73] Latendresse, M. Scalable real-time weather server written in Scheme. In *International Lisp Conference*, October 2002.
- [74] Lindholm, T. and F. Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Addison-Wesley, April 1999.
- [75] Meunier, P., R. B. Findler, P. Steckler and M. Wand. Selectors make analysis of case-lambda too hard. In *Scheme and Functional Programming*, 2001.
- [76] Meyer, A. R. and J. G. Riecke. Continuations may be unreasonable. In *ACM Conference on LISP and Functional Programming*, pages 63–71, 1988.
- [77] Microsoft Corporation. <http://www.microsoft.com/net/>.
- [78] Miller, S. G. SISC: A complete Scheme interpreter in Java, February 2003.
<http://sisc.sourceforge.net/>.
- [79] Milner, R., M. Tofte, R. Harper and D. MacQueen. The denition of Standard ML (revised), 1997.
- [80] Mogul, J. The case for persistent-connection HTTP. In *Computer Communication Review*, pages 299–313, October 1995.
- [81] Myers, B., S. E. Hudson and R. Pausch. Past, present and future of user interface software tools. In Carroll, J. M., editor, *HCI In the New Millennium*, pages 213–233. ACM Press, Addison-Wesley, 2001.

- [82] NCSA. The common gateway interface. <http://hoohoo.ncsa.uiuc.edu/cgi/>.
- [83] Neubauer, M. and P. Thiemann. Type classes with more higher-order polymorphism. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.
- [84] Open Market, Inc. FastCGI specification. <http://www.fastcgi.com/>.
- [85] Pai, V. S., P. Druschel and W. Zwaenepoel. IO-lite: A unified I/O buffering and caching system. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.
- [86] Pierce, B. C. *Types and Programming Languages*. MIT Press, 2002.
- [87] Pitman, K. Special forms in Lisp. In *Conference Record of the Lisp Conference*, August 1980. Stanford University.
- [88] Plotkin, G. Call-by-name, call-by-value, and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [89] Plotkin, G. D. A structural approach to operational semantics. Technical Report Technical Report FN-19, Aarhus University, Computer Science Department, 1981.
- [90] Postel, J. B. RFC 821: Simple mail transfer protocol, August 1982.
<http://www.ietf.org/rfc/rfc0821.txt>.
- [91] Queindec, C. The influence of browsers on evaluators or, continuations to program Web servers. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.
- [92] Reed, D. P. Implementing atomic actions on decentralized data. In *ACM Transactions on Computer Systems*, pages 234–254, February 1983.
- [93] Rémy, D. Typechecking records and variants in a natural extension of ML. In *ACM Symposium on Principles of Programming Languages*, 1989.
- [94] Reynolds, J. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Conference*, volume 2, pages 717–740. ACM, August 1972.
- [95] Roth, M. and E. Pelegrí-Llopert. Javasever pages specification, August 2002.
<http://java.sun.com/products/jsp/>.

- [96] Sabry, A. *The Formal Relationship between Direct and Continuation-passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Rice University, 1994.
- [97] Sabry, A. and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 1993.
- [98] Sakamoto, T., T. Sekiguchi and A. Yonezawa. Bytecode transformation for portable thread migration in Java. In *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents*, pages 16–28, September 2000.
- [99] Sandholm, A. and M. I. Schwartzbach. A type system for dynamic Web documents. In *Symposium on Principles of Programming Languages*, pages 290–301, 2000.
- [100] Spoonhower, D., G. Czajkowski, C. Hawblitzel, C.-C. Chang, D. Hu and T. von Eicken. Design and evaluation of an extensible Web and telephony server based on the J-Kernel. Technical report, Department of Computer Science, Cornell University, 1998.
- [101] Steele, Guy L., J. Rabbit: A compiler for Scheme. Master’s thesis, MIT, 1978.
- [102] Steele Jr., G. L. *Common Lisp: The Language, 2nd Edition*. Digital Press, 1990.
- [103] Strachey, C. and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps, technical monograph PRG-11. Technical report, Oxford University Computing Laboratory, Programming Research Group, 1974.
- [104] Sun Microsystems. Object serialization specification.
<ftp://ftp.java.sun.com/docs/j2se1.3/serial-spec.pdf>.
- [105] Sun Microsystems, Inc.
<http://java.sun.com/products/jpda/>.
- [106] Sun Microsystems, Inc. Forte tools.
<http://www.sun.com/forte/>.
- [107] Thau, R. Design considerations for the Apache server API. In *Fifth International World Wide Web Conference*, May 1996.

- [108] Thielecke, H. Comparing control constructs by typing double-barrelled CPS transforms. In *Third ACM SIGPLAN Workshop on Continuations*, 2001.
- [109] Thiemann, P. WASH/CGI: Server-side Web scripting with sessions and typed, compositional forms. In *Practical Applications of Declarative Languages*, 2002.
- [110] van Rossum, G. Python reference manual. Technical report, Corporation for National Research Initiatives, October 1996.
- [111] Wall, L. and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1992.
- [112] Wand, M. Continuation-based multiprocessing. In *ACM Conference on LISP and Functional Programming*, pages 19–28, 1980.
- [113] Wand, M. Complete type inference for simple objects. In *IEEE Symposium on Logic in Computer Science*, June 1987.
- [114] Wileden, J. C., A. Kaplan, G. A. Myrestrand and J. V. Ridgway. Our SPIN on persistent Java: The JavaSPIN approach. In *First International Workshop on Persistence and Java*, September 1996.
- [115] Williams, N. J. An implementation of scheduler activations on the NetBSD operating system. In *USENIX Annual Technical Conference*, 2002.
- [116] Wright, A. K. and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
- [117] Zandy, V. and B. Miller. Reliable network connections. In *ACM International Conference on Mobile Computing and Networking*, pages 95–106, September 2002.
- [118] Zandy, V. and B. Miller. Checkpoints of gui-based applications. In *USENIX Annual Technical Conference*, June 2003.

Appendix A

Expressiveness of *send/suspend*

Scheme's *call/cc* construct, can easily result in confusing programs [50, 76]. Programming with *send/suspend*, on the other hand, seems straight forward. This raises several interesting questions. Why does *send/suspend* seem simpler? Is *send/suspend* less powerful (or less expressive [40]) than *call/cc*? If *send/suspend* provides only a restricted version of *call/cc* then perhaps there is a more efficient way to implement *send/suspend*.

Addressing these questions first requires a meaningful comparison between *send/suspend* and *call/cc*. Superficially, the two primitives behave very differently. In particular, *send/suspend* interacts with the consumer, who chooses both which form to submit and the values of the form's fields. The *call/cc* construct, however, gives the program control of which continuation to resume and the continuation's argument.

Removing this obvious distinction between *call/cc* and *send/suspend* yields a more meaningful comparison. To assist with this goal, a hypothetical browser removes the human consumer from the system. This browser always immediately submits any form it receives using the program-supplied initial field values. In practice, this hypothetical browser is easily implemented by including a line of ECMAScript (Javascript) [38] in each Web form.

With *send/suspend*'s behavior completely under the program's control, does *send/suspend* provide enough expressive power to implement *call/cc*? It seems like *send/suspend* may provide less flexibility because it both captures and invokes a continuation while *call/cc* separates the

capturing and invoking operations. Thankfully, this is not actually a problem. By invoking *send/suspend* in an unusual manner, the program can render one of the actions of capturing or invoking a continuation useless, thus leaving only the desired action.

Initially defining a function *cawl-cc* to be *send/suspend* shows how *cawl-cc* is not quite *call/cc*. Refining this first attempt in a reasonably systematic manner leads to the desired implementation.

```
(define cawl-cc send/suspend)
```

Since the type of *send/suspend* does not match that of *call/cc*, the above definition clearly does not work. The type of *call/cc* is $((\alpha \rightarrow \text{answer}) \rightarrow \alpha) \rightarrow \alpha$ while the type of *send/suspend* is $(\text{url} \rightarrow \text{response}) \rightarrow \text{request}$.

To obtain enough space to correct the type mismatch, the next step is to η expand the definition. Although η expansion is not always sound in the call-by-value lambda calculus due to the altered evaluation order and the possibility of replacing a non-function with a function, this step is semantics preserving assuming that *send/suspend* is a bound to a function and its binding is not mutated.

```
(define (cawl-cc f) (send/suspend f))
```

The argument to *send/suspend* must be a function that consumes a URL, but *f* consumes a continuation instead. Again, η expansion provides room to work. Again this step does not change the program assuming *f* is bound to a function.

```
(define (cawl-cc f)
  (send/suspend
    (lambda (k-id)
      (f k-id))))
```

Now the problem is that *f* receives a URL rather than a continuation. Again η expansion provides procrastination. Since *k-id* is not a function, this slightly changes the program's behavior—an error will not occur until the η expansion of *k-id* is invoked.

```
(define (cawl-cc f)
  (send/suspend
    (lambda (k-id)
```

```
(f (λ (x)
    (k-id x))))))
```

The next type-error manifests itself in the application of *k-id* to *x*. The only mechanism for invoking the continuations associated with URLs is hidden somewhere inside *send/suspend*.

Therefore, the function passed to *f* calls *send/suspend* a second time.

```
(define (cawl-cc f)
  (send/suspend
   (λ (k-id)
     (f (λ (x)
         (send/suspend
          (λ (unused-k-id)
            (k-id x))))))))))
```

The second *send/suspend* discards the unused continuation it (inadvertently) captures and sends the old *k-id* to the browser to be re-invoked. The expression *(k-id x)* is still causing problems. Based on the return type of the argument to *send/suspend*, the expression *(k-id x)* should be a Web form. Thus, the **form** function create a Web form (actually a *response*) that sends *x* as a payload to the (continuation associated with the) URL *k-id*. Since the browser automatically submits forms with the default values, this happens at once.

```
(define (cawl-cc f)
  (send/suspend
   (λ (k-id)
     (f (λ (x)
         (send/suspend
          (λ (unused-k-id)
            (form k-id '([payload ,x]))))))))))))
```

The form carrying the payload *x* is the result of the first *send/suspend*, which captured the continuation initially. The return type of *cawl-cc*, however, should be the same as *x*, not necessarily a form (actually a *request*). Before returning, *cawl-cc* must extract the *payload* field.

Since the second call to *send/suspend* never returns, no extraction is needed.

```
(define (cawl-cc f)
  (form-field
   'payload
   (send/suspend
    (λ (k-id)
      (f (λ (x)
```

```
(send/suspend
  (λ (unused-k-id)
    (form k-id '([payload ,x]))))))))
```

The code above should work fine if f always invokes its continuation argument. The problem arises in the case when f returns normally. In this case, the usual trick [63] of explicitly applying the continuation to the value returned by f suffices.

```
(define (cawl-cc f)
  (form-field
   'payload
   (send/suspend
    (λ (k-id)
      (let ([k (λ (x)
                  (send/suspend
                   (λ (unused-k-id)
                     (form k-id '([payload ,x])))))]
                (k (f k))))))))))
```

The above code type-checks and appears to behave properly when tested. This sequence of code transformations starts with a trivial definition. Although incorrect, the initial code embodies roughly the right idea—that *call/cc* and *send/suspend* can express similar control flows. Following the types in a reasonably naïve manner¹ lead to an implementation of *call/cc*.

For further improvement, *send/back* can replace the call to *send/suspend* that ignores the captured continuation.

```
(define (cawl-cc f)
  (form-field
   'payload
   (send/suspend
    (λ (k-id)
      (let ([k (λ (x) (send/back (form k-id '([payload ,x])))))]
                (k (f k))))))))))
```

Passing the “wrong” k -url to *send/suspend* is probably not necessary. Instead, sending messages to control the browser’s backtracking would work, assuming the browser actually kept all old continuations. This is essentially the same.

Figure A.1 implements the **form** and **form-field** functions used by *cawl-cc*. The **form** function creates an HTML form containing hidden fields, a submit button, and one line of EC-

¹A few steps involved some semantic considerations such as the intention of throwing the k -id URL as a continuation.

```

;; : str[url] (listof (list sym str)) → response
;; to build a Web page that automatically submits itself using javascript
(define (form k-url hidden-fields)
  `(html (head (title "Auto-form "))
    (body (form ([action ,k-url] [method "post"] [name "the_form"])
      (input ([type "submit"] [name "the_submit_button"]])
        . ,(map (λ (x)
          `(input ([type "hidden"]
            [name ,(symbol→string (π0 x))]
            [value ,(marshal (cadr x))]))
            hidden-fields))
        (script "document.the_form.the_submit_button.click()")))))

;; : sym request → str
(define (form-field key form)
  (unmarshal (extract-binding/single key (request-bindings form))))

;; marshal : (tst → pickle)
;; unmarshal : (pickle → tst)
(define-values (marshal unmarshal)
  (let ([pickles (make-hash-table)])
    (values
      (λ (x)
        (let* ([key-str (symbol→string (gensym))]
          [key (string→symbol key-str)]
          (hash-table-put! pickles key x)
          key-str))
        (λ (y)
          (hash-table-get pickles (string→symbol y)))))))

```

Figure A.1: Form Functions For Cawl-cc

MAScript, "document.the_form.the_submit_button.click()", that automatically returns the form to the server. Since the continuations produced by *cawl-cc* must accept arbitrary Scheme values, these values must be marshaled before traveling over the network in a form. The *marshal* function, despite its name, does not quite marshal the data since several types of opaque Scheme values such as closures, continuations, threads, I/O ports, and some structures do not marshal easily. Instead, *marshal* stores its argument in a hash table and returns the key.