

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

NOTE TO USERS

This reproduction is the best copy available.

UMI

RICE UNIVERSITY

**Programming the Web with
High-Level Programming Languages**

by

Paul Graunke

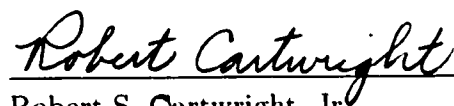
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Masters of Science

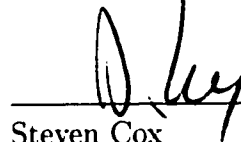
APPROVED, THESIS COMMITTEE:



Matthias Felleisen
Professor of Computer Science, Co-chair



Robert S. Cartwright, Jr.
Professor of Computer Science, Co-chair



Steven Cox
Professor of Computational and Applied
Mathematics

Houston, Texas

April, 2001

UMI Number: 1405668



UMI Microform 1405668

Copyright 2001 by Bell & Howell Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Programming the Web with High-Level Programming Languages

Paul Graunke

Abstract

Concepts from high-level languages can greatly simplify the design and implementation of CGI programs. This dissertation develops two systems for implementing these programs.

The first technique* relies on a custom Web server that dynamically loads CGI programs using the operating system-style services of MrEd, an extension of Scheme. The server implements programming mechanisms using continuations that allow the CGI program to interact with the user in a natural manner.

The second technique relies on program transformations from functional language compilation.[†] It allows the use of standard servers and alleviates most of the memory consumption on the server.

In my thesis I discuss the advantages and disadvantages of each approach. I conclude with suggestions for further investigations into this topic.

*The first technique previously appeared at the European Symposium on Programming, 2001 A.D., in a paper with the same title as this dissertation, coauthored with Shriram Krishnamurthi, Steve van der Hoeven, and Matthias Felleisen.

[†]The compilation based technique was submitted to the International Conference on Functional Programming in a paper titled, "How to Design and Generate CGI Programs: Applying Functional Compiler Techniques to the Real World," coauthored with Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen.

Acknowledgments

I could not have earned a masters degree in computer science without support from many instrumental people.

Shriram Krishnamurthi ran the lab session for my first programming course at Rice, where I discovered Scheme and the view of programming as an exercise in mathematical elegance. As a graduate student, he taught me about compilation of advanced languages, and more importantly how to distill and present a slew of papers. Later Shriram introduced me to Web programming, where I saw the practical need for something better. His ability to highlight the significance of ideas motivated me to pursue several topics he suggested, resulting in this dissertation. In many ways Shriram acted as an advisor, though I was not able to officially add him to my committee.

Matthias Felleisen, paternal taskmaster of the PLT, steered me in the right direction and kept me focused by pruning unnecessary tasks off of my work list. His dedication and constant availability rescued me from many tight spots along the way.

Corky Cartwright welcomed me into the fold and patiently endured my early floundering. I admire his consistently desiring the best for others, even when I switched to this topic.

Matthew Flatt, whose software and ideas the rest of PLT builds upon, proved that graduate students can work twelve hour days and still watch television and golf.

Robby Findler not only provided insight on better designs, but also gave new meaning to “lunch”.

Both of my parents emphasized the importance of education and consistently went the extra mile that I should have the best. My father kindled my interest in

programming early on and taught me to learn from books independently.

Vineyard Christian Fellowship sustained me through many a doubt and times of insecurity. The pastors, Michael, Anthony, Alan, Chad, Shae, Bill, and Leighnae, numerous friends from homegroups, ministry team members, and others often renewed my hope by their friendship, encouragement, and prayers.

More than any other, I'm eternally indebted to Jesus Christ, who died to reconcile me to God and who rose again to break the sting of death. His Holy Spirit lifted me up in lonely times, surprised me with peace in the midst of anxiety, and valued me in the midst of failures.

Contents

Abstract	ii
Acknowledgments	iii
List of Illustrations	vii
1 Introduction	1
1.1 Web Servers and High-Level Operating Systems	2
1.2 Designing CGI Programs	3
2 Web Servers are Operating Systems	6
2.1 MrEd: a High-Level Operating System	6
2.2 Serving Static Content	8
2.2.1 Implementation of the Web Server's Core	8
2.2.2 Performance	9
2.3 Dynamic Content Generation	11
2.3.1 Simple Dynamic Content Generation	11
2.3.2 Content Generators are First-Class Values	13
2.3.3 Exchanging Values between Content Generators	13
2.3.4 Interactive Generation of Content	14
2.3.5 Implementing Interactive Generation of Content	17
2.3.6 Performance	18
2.4 Modularity of the Server	19
3 Compiling CGI Programs	21
3.1 CGI Programs	21

3.2	Generating CGI Programs	24
3.3	Direct-Style CGI Programs	24
3.4	Functional CGI Programs	26
3.5	Compiling Stateful CGI Programs	30
3.6	Developing CGI Scripts	32
3.7	Implementation Status	34
4	Related Work	38
5	Future Work	43
6	Conclusions	45
	Bibliography	47

Illustrations

2.1	MrEd's TCP, thread and custodian primitives	7
2.2	Performance for static content server	10
2.3	Additional content generator primitives	15
2.4	An interactive CGI program	16
2.5	Performance for dynamic content generation	19
3.1	An Interactive Addition Program	22
3.2	An Equivalent CGI Version	22
3.3	The Compiled Version of Figure 3.1	25
3.4	The CGI Version	27
3.5	A Stateful Interactive Program	30
3.6	A Stateful CGI Version	35
3.7	CGI Error Reporting	36
3.8	CGI Stepping	37

Chapter 1

Introduction

The Web's growing significance as a distributed computing platform forces us to understand its underlying computational nature. Some researchers study the nature of data on the Web, especially their type systems. We conduct research on control aspects for Web programs. Here we focus on the Common Gateway Interface (CGI) [35] protocol for generating dynamic Web content.¹

The need for generating Web information on demand is obvious. One page may need the current time and date; another page may display the current status of the server; and a third page may include results from a database query. Since such CGI programs compute small pieces 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, customer interviews, interactive games, and more implemented via the CGI interface. In other words, instead of writing CGI *scripts*, programmers now design, implement, and maintain interactive CGI *programs* with complex and multi-layered interface protocols.

The designers of complex, interactive CGI programs face a serious design problem. CGI programs halt after producing the first Web page. Most dialogs, however, consist of many interactions, and each interaction presents a form and processes a response. Thus, to reconcile the interactive nature of programs with the CGI standard,

¹Our work applies to Java servlets as well.

an interaction is implemented by having a script deliver a Web page, wait for the consumer to submit a response, and process the response with a(nother) CGI script. Complicating matters even more, the CGI 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 CGI program and a consumer make up a pair of multiply resumable coroutines, but due to the lack of coroutines or similar constructs in most Web scripting languages, the designer cannot match the structure of the program to the structure of the interaction.²

Hughes [27] and Queinnec [40] recognized this structure clash problem. They suggest a solution based on the (more or less explicit) manipulation of continuations. That is, when a CGI script queries a consumer, it grabs the current continuation and preserves it for future uses. When a consumer submits answers to a form query, the server resumes the continuation with the bindings from the form. Due to the capabilities of Web browsers, a consumer may also backtrack in an interaction, and thus trigger several invocations of one and the same continuation.

1.1 Web Servers and High-Level Operating Systems

One way to implement additional language constructs that ease the development of interactive CGI programs involves a custom Web server. The server dynamically loads code into the server itself. Since the underlying operating system no longer manages the CGI programs, the server must compensate.

The Web server provides operating system-style services. Like an operating system, the server runs programs (e.g., CGI scripts). Like an operating system, the server protects these programs from each other. And, like an operating system, the server manages resources (e.g., network connections) for the programs it runs.

²Java servlets have a similar problem. They contain a single method for GET or POST requests, and cannot wait for additional submissions or resubmissions from the consumer.

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. The second chapter of this dissertation 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 CGI 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 [8], this performance improvement is important. Finally, our server provides programming mechanisms for the dynamic generation of Web content that are difficult to support in a conventional server architecture.

The basis of our experiment is MrEd [18], an extension of Scheme [30]. The implementation of the server heavily exploits four extensions: first-class *modules*, which help structure the server and represent server programs; 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. The second chapter also shows which role each construct plays in the construction of the server.

1.2 Designing CGI Programs

Although building a custom Web server yields high performance and supports a more natural style of programming, it has several drawbacks. The third chapter addresses these problems by examining the process of writing CGI programs by hand, which leads to a series of transformations that compile interactive programs into CGI programs automatically.

If the implementation language of a CGI program supports first-class continuations—such as Scheme's *call/cc* or Haskell's continuation monads—the design and implemen-

tation of a CGI program reduces to the design and implementation of a sequential program whose interactions with the consumer are implemented with continuations. As Hughes points out in his work on arrows, a CGI program grabs a continuation, produces a textual representation, sends this representation to the client (in a hidden form field), and waits for the client to resume the dialog. Few languages support this kind of operation on continuations, however.

Worse, the continuation solution leaves open the problem of where to place the stores of (internally) imperative CGI programs. Placing the store on the server and providing (symbolic) pointers to these values to the consumer creates a problem of distributed garbage collection where remote pointers may be in bookmark files, human minds, and other media. It also makes the consumer depend on the reliability of the server. Monads (or arrows) also do not solve the problem. They turn the store into a lexical variable that is threaded through the program. When an arrow-based CGI script suspends its operation, it stores the lexical variables of the continuation in a hidden field [27]. Thus, if a consumer clones a browser window, the store is copied, which introduces coherence problems. In short, the problem is how to provide the power of, say, continuation operations to the designer of CGI programs in all languages and how/where to save the store of a suspended program.

The third chapter of this dissertation describes how CGI programmers can use *existing* design methods to develop interactive programs and that well-known, automatable transformations can *generate* CGI scripts from these programs. Specifically, we extend a programming language with a primitive for CGI interactions and show how this extension simplifies the design and development of interactive CGI programs; how it allows programmers to migrate legacy programs to the Web; how our implementation addresses the store problem; and how we can adapt existing programming environments in support of this development style.

We have implemented our ideas in Scheme, so that we can test each development stage. The work does not rely on Scheme's continuation mechanism, however. We

discuss in the last chapter how our ideas carry over to all kinds of programming languages and how they are useful even in the absence of tool support.

The fourth chapter relates this dissertation to the work of others. The fifth chapter proposes future directions for further research. The final chapter concludes with a summary.

Chapter 2

Web Servers are Operating Systems

The following section is a brief introduction to MrEd. Section 2.2 explains the core of our server implementation. In section 2.3 we show how the server can be extended to support Scheme CGI scripts and illustrate how programming in the extended Scheme language facilitates the implementation of scripts. Sections 2.2 and 2.3 also present performance results. Section 2.4 discusses the internal structure of the server.

2.1 MrEd: a High-Level Operating System

MrEd [18] is a safe implementation of Scheme [30]; it is one of the fastest existing Scheme interpreters. 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 modules. The module system [17] permits programmers to specify *atomic units* and *compound units*. An atomic unit is a closed collection of definitions. Each unit has an import and an export signature. The import signature specifies what names the module 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

```

tcp-listen : Nat [Nat] → 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 →* Input-port Output-port
;; creates I/O ports for a connection request via the listener

```

```

thread : (→ Void) → Thread
;; spawns a thunk as a thread

make-semaphore : Nat → Semaphore
;; creates a semaphore with specified number of tokens

semaphore-post : Semaphore → Void
;; posts a semaphore and releases waiting threads

semaphore-wait : Semaphore → Void
;; waits (and possibly suspends) for a semaphore

```

```

make-custodian : → Custodian
;; creates a custodian

custodian-shutdown-all : Custodian → Void
;; shuts down all threads in custodian and reclaims all resources

```

Figure 2.1 : MrEd's TCP, thread and custodian primitives

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) 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 2.1 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 ([*parameter1 value1*]...) *body1* ...)

sets *parameter1* to *value1* for the dynamic extent of the computation *body1* ...; 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 Scheme system. Setting this parameter to another function that raises an exception or performs clean up operations will manage calls to *exit* appropriately.

Finally, MrEd provides a mechanism for managing resources, such as threads (with associated parameter bindings), 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 2.1 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.

2.2 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 section explains the core of our server implementation. The second section compares performance figures of our server to Apache [2], a widely-used, commercially-deployed server.

2.2.1 Implementation of the Web Server's Core

The core of a Web server is a wait-serve loop. It waits for requests on a particular TCP port. For each request, it creates a thread that serves the request. Then the

server recurs:¹

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

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.²

2.2.2 Performance

It is easy to write compact implementations of systems with high-level constructs, but we must demonstrate that we do not sacrifice performance for abstraction. More precisely, we would like our server to serve content from files at about the same rate as Apache [2]. We believed that this goal was within reach because most of the

¹**let-values** binds names to the values returned by multiple-valued computations such as *tcp-accept*, which returns input and output ports.

²The server may actually loop to handle multiple requests per connection. This dissertation does not explore this possibility further.

	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 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 2.2 : Performance for static content server

computational work involves parsing HTTP requests, reading data from disk, and copying bytes to a (network) port.³

To verify this conjecture, we compared our 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 2.2 show that we have essentially achieved our goal. The results were obtained using the S-client measuring technology [5]. For the historically most common case [4]—files between six and thirteen kB—our server performs at a rate of 60% to 80% of Apache. For larger files, which are now more common due to

³This assumes that the server does not have a large in-memory cache for frequently-accessed documents.

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 CPU cycles. For the larger 100 kB files, both servers drove the network card at full capacity.

2.3 Dynamic Content Generation

Over the past few years, the Web's content has become increasingly dynamic. *USA Today*, for instance, finds that as of the year 2000 A.D., more than half of the Web's content is generated dynamically [8]. 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 2.2 accommodate dynamic extensions, that the performance of the revised server is superior to that of Apache,⁴ and that it supports a new programming paradigm that is highly useful in the context of dynamic Web content generation.

2.3.1 Simple Dynamic Content Generation

Since a single server satisfies different requests with different content generators, we implement a generator as a module that is dynamically invoked and linked into the server context. More specifically, a CGI program in our world is a unit:

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

It exports nothing; it imports the names specified in the *cgi^* signature. The result of its final expression (and of the unit invocation) is an HTML page.⁵

Here is a trivial CGI program using *quasiquote* [39] to create an HTML page with *unquote* (a comma) allowing references to the *TITLE* definition.

⁴Apache outperforms most other servers for CGI-based content generation [1].

⁵To be precise, it generates an X-expression, which is an S-expression representation of an XML document. The server handles other media types also; we do not discuss these in this dissertation.

```
(unit/sig () (import cgi^)
  (define TITLE "My first Web page")
  `(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 imports of a content generator supply the request method, the URL, the optional headers, and the bindings:

```
(define-signature cgi^ (
  method      ; (union 'get 'post)
  url         ; Url
  headers     ; (listof (cons Symbol String))
  bindings    ; (listof (cons Symbol String))
  ...))
```

To add dynamic content generation to our server, we modify the *dispatch* function from section 2.2 to redirect requests for URLs starting with `"/cgi-bin/"`. More concretely, instead of responding with the 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 (cgi-url? url)
  (let ([cust (make-custodian)])
    (parameterize
      ([current-custodian cust]
       [exit-handler (lambda (x) (custodian-shutdown-all cust))])
      (let ([cgi-program (cached-load (url-path url))])
        (output-xhtml (invoke-unit/sig cgi-program cgi^)))))
  ...)
```

The newly installed custodian is shut down on termination of the CGI script. 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 not the entire server.

2.3.2 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 cgi^)
    (set! count (add1 count))
    `(html (head (title "Testing Persistent State of Counter"))
      (body (p "This is a cgi 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*.

2.3.3 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 our server architecture, dynamic content generators can naturally exchange high-level forms of data through the common heap.

Our dynamic content generation model features a simple extension that permits multiple generators to be defined within a single lexical scope. The current unit of granularity in the implementation is a file. That is, one file may yield an expression that contains multiple generators. The expression may perform arbitrary operations to define and initialize the shared scope. To distinguish between the generators, the server's interface requires that the file return an association list of type

```
(listof (cons Symbol Content-generator))
```

For instance, a file contain two content generators:

```
(let* ([data-file ...]
      [lock (make-semaphore 1)])
  ((' (add . , <Content-generator>1)
    (delete . , <Content-generator>2))))
```

This yields two generators that share a lock to a common file. The distinguishing name in the association is treated as part of the URL in a CGI request.

2.3.4 Interactive Generation of Content

Christian Queinnec suggested in his ICFP article [40], that Web browsing in the presence of dynamic Web content generation can be understood as the process of capturing and resuming continuations.⁶ 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 content generator'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-CGI programs that are being refitted with Web-based interfaces. To support this common programming paradigm, our server links content generators to the three additional primitives in figure 2.3.

The *send/suspend* function allows the content generator to send an HTML form to the client for further input. The function captures the continuation and suspends the computation of the content generator. When the user responds, the server resumes

⁶This idea also appears in Hughes's paper on arrows [27].

```

send/suspend : (Url → Html-page) →* Method
                                     Uri
                                     (listof (cons Symbol String))
                                     (listof (cons Symbol String))

send/finish : Html-page → Void
adjust-timeout : Nat → Void

```

Figure 2.3 : Additional content generator primitives

the continuation with four values: the request method, the URL, the optional headers, and the form bindings.

To implement this functionality, *send/suspend* consumes a function of one argument. This 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 2.4, which presents a simple example of an interactive content generator. This script implements curried 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 content generator 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

```

(unit/sig () (import cgi))

;; get-input-w/-short-form : String → (String → Html-page)
(define (get-input-w/-short-form which-one)
  (let-values (((method url headers bindings)
                (send/suspend
                 (lambda (k-url)
                   '(html (head (title ,which-one " number"))
                        (body
                          (form ((method "post") (action ,k-url))
                                "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
...

'(html (head (title "Product"))
      (body (p "The product is: "
               ,(string-multiply (get-input-w/-short-form "first")
                                (get-input-w/-short-form "second"))))))

```

Figure 2.4 : An interactive CGI program

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.

Based on this analysis, our server implements the following policy. When the content generator returns or calls the *send/finish* primitive, all the continuations associated with this generator computation are released. Generators that wish to keep the continuations active can suspend instead. When a user attempts to access a reclaimed continuation, a page directs them to restart the computation. Furthermore,

each instance of a content generator 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 *adjust-timeout*. This mechanism for shutting down the generator only works because the reaper and the content generator's thread share the same custodian. This illustrates why custodians, or resource management in general, cannot be identified with individual threads.

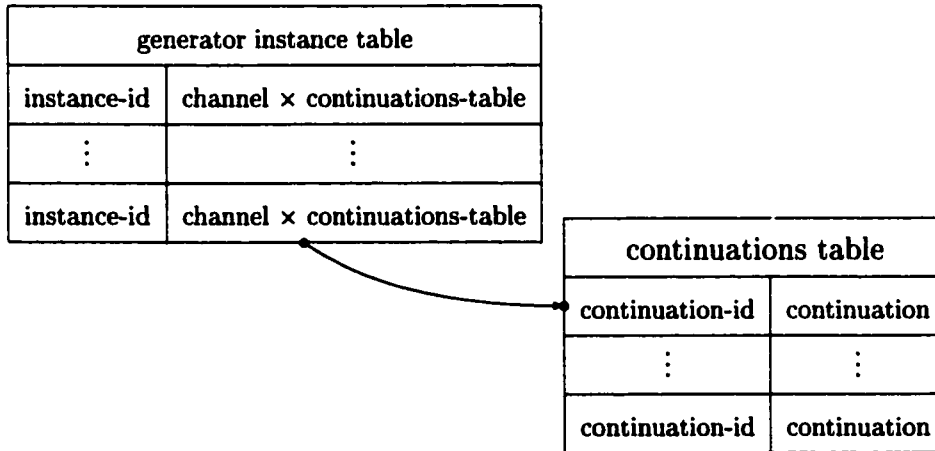
2.3.5 Implementing Interactive Generation of Content

The implementation of the interactive CGI policy is complicated by the (natural) MrEd 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 content generator, 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/cgi-bin/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 content generator actually has two tiers. The first tier associates instance identifiers for content generators 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 generator's continuation, it looks up the content generator in one table, and extracts the channel and continuation table for that generator. The server then looks up the desired continuation in this second table and passes it through the channel to the suspended thread along with the request information and the ports for the current connection.

The two-tier structure of the tables also facilitates clean-up. When the time limit for an instance of a content generator expires or when the content generator computation terminates, the instance is removed from the generator instance table. This step, in turn, makes the continuation instance table inaccessible and thus available for garbage collection.

2.3.6 Performance

We expect higher performance from our Web server than from conventional servers that use the Common Gateway Interface (CGI) [35]. A conventional server starts a separate OS process for each incoming request, creating a new address space, loading code, etc. Our server eliminates these costs by avoiding the use of OS process boundaries and by caching CGI programs.

Our experiments confirm that our server handles more connections per second than CGI programs written in C. For example, for the comparison in figure 2.5, we clock 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 our server scale much better relative to response size than FastCGI does. We conjecture that this is because FastCGI copies 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.

	CGI		FastCGI		MrEd Full		MrEd Lite	
Clients	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 2.3.4, 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 2.5 : Performance for dynamic content generation

2.4 Modularity of the Server

Web servers must not only be able to load Web programs (e.g., CGI scripts) but also load new modules in order to extend their capabilities. For example, requiring password authentication for specific URLs affects serving content from files and from all dynamic content generators. 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 [46] allows the builder of the Web server to replace pieces of the server's response process, such as those outlined above, with their own code. The builder installs structures with function pointers into a Chain of Command pattern [21]. 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, our server is constructed in a completely modular fashion using the unit system [17]. This provides the flexibility of the Apache module system in a less ad hoc, more hierarchical manner. To replace part of how the server responds to requests, the server builder writes a compound unit that links the provided units and the replacement units together, forming an extended server. Naturally, the replacement units may link in the original units and delegate to them as desired.

Using units instead of dynamic protocols has several benefits. First, the server does not 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 are subject to the same level of static analysis as the original server. (Our soft-typing tool [16] revealed several easily corrected premature end-of-file bugs in the original server.)

Chapter 3

Compiling CGI Programs

3.1 CGI Programs

A typical interactive program performs a series of computations interspersed with interactions with the user. An interaction presents a request for information and waits for the user's response. When the computation completes, the program produces the final result. Figure 3.1 presents a trivial interactive program that requests two numbers, adds them, and displays the result.

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 pieces. 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 clone functionality of a browser, the consumer can submit two responses to a 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 accomplish this is to separate the program into several pieces, one per interaction and one for the last step. When a piece has finished its task, the execution stops. All information from one piece of the program required by some other piece must be explicitly turned over to the next piece. There are several different

```

;; prompt-read : String → Value
;; read an S-expression as a Scheme value
(define (prompt-read question)
  (display question)
  (read))

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

Figure 3.1 : An Interactive Addition Program

```

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

(define FIRST-STOP "first number done")

(define SECOND-STOP "second number done")

(define bindings (get-bindings))

(cond
  [(null? bindings)
   (produce-html "Enter the first number to add:" FIRST-STOP '())]
  [(string=? (extract-binding/single 'cont bindings) FIRST-STOP)
   (produce-html "Enter the second number to add: " SECOND-STOP
                '((first-number ,(extract-binding/single 'response bindings))))]
  [(string=? (extract-binding/single 'cont bindings) SECOND-STOP)
   (show (+ (string→number (extract-binding/single 'first-number bindings))
            (string→number (extract-binding/single 'response bindings))))])
)
```

Figure 3.2 : An Equivalent CGI Version

methods, but in each case the data is marshaled into a string and placed in a hidden HTML field, a cookie, or into a file on the server.

Figure 3.2 shows the addition program converted into a CGI program. Because the former contains two interactions, the latter consists of three pieces, re-integrated into a single program via a conditional. Technically, the CGI program gets the bindings from the Web form and then tests three conditions:

1. If there are no bindings, the program is called without inputs. It creates a Web page with a question, a hidden field that specifies the current continuation, and the list of values that are supposed to be hidden in the Web page.
2. If the program can extract a binding for 'cont, 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 a different binding for 'cont, 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” using the back button. 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.

¹M. Jackson [28] 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 a another file, it is best to think of the program as two coroutines. Since COBOL does not support coroutines, he invented *program inversion*, a technique for providing simple coroutine-like procedures in programs that do not support such forms of control.

Still, 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 produce their scripts in an ad hoc fashion, we propose that the development process should take advantage of this relationship. We show in the following sections how this can be done.

3.2 Generating CGI Programs

Programmers 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 design methods for interactive programs for this chaotic world of Web programming. In this section, we first explain briefly how we can accomplish this goal with a custom Web server and what the problems of this solution are. Second, we explain how we can use this idea as the starting point for the development of a pre-processor that restructures interactive programs into CGI programs.

3.3 Direct-Style CGI Programs

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 continuation, using *call/cc*, and can store this continuation in the server.

We have implemented this approach, as has Queinnec [40]. The previous chapter demonstrates that this approach has distinct advantages. Most importantly, the modified Web server yields superior speed for CGI scripts compared to several existing methods.

Unfortunately, the approach has three severe problems. First, it requires a server written in a language with advanced control features such as continuations. Second, the programmer must modify existing CGI programs to use the server’s new and non-standard CGI interface. Third, the URLs for continuations act as persistent

```

(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 (apply (vector-ref closures (closure-code f)) (closure-env f)) args))

;; the converted functions and continuations
(define closures
  (vector
    (lambda ()
      (lambda (response1)
        (prompt-read-k "Enter the second number to add:"
          [make-closure 1 (list response1)]))))

    (lambda (response1)
      (lambda (response2)
        (display (+ response1 response2))))))

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

(prompt-read-k "Enter the first number to add:" [make-closure 0 '()])

```

Figure 3.3 : The Compiled Version of Figure 3.1

references to storage within the server. This results in a distributed garbage collection problem with no support from the browser. 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, timeouts do not solve the problem. 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.

Our experience using the server for the TeachScheme! project's registration program highlighted the problem of timeouts further. A timeout of 24 hours sufficed for most teachers; 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, i copied the first page generated by the registration program to a different file, referenced by a different URL. Testing indicated that the copied page worked, yet during the next day several TeachScheme! administrators indicated otherwise. Even though none of the code nor static pages changed, the links ceased to function due to the timeout.

3.4 Functional CGI Programs

The key problem with our custom Web server is that its CGI scripts critically rely on Scheme's capability to capture a continuation with *call/cc*. To eliminate the uses of *call/cc*, we turn to techniques for compiling functional programming languages. More specifically, we employ three well-known transformations:

The Continuation Passing Style [20] eliminates *call/cc* by representing the control state of a program explicitly. In particular, each function of the program now consumes one additional argument: the continuation function. A function that must grab the continuation and store it for future uses can therefore just refer to this new argument. In our 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 turns the resumption points into independent functions that can be moved to the top level.

Closure conversion changes the representation of closures into a first-order form. Using this new form, the script can write a continuation closure into a hidden field of a Web form and use it later to restart a computation.

Once we have CPS'ed, lambda lifted, and closure-converted an interactive program, we can turn it into a CGI script by adding some primitives and replacing others.

```

(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 3.3

(define closures ...) ; as in figure 3.3

;; 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-compatible HTTP header and HTML Web form
(define (produce-html question mark free-values)
  ...)

(define bindings (get-bindings))

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

```

Figure 3.4 : The CGI Version

We explain the process with the trivial but illustrative example from figure 3.1. The result of the three compilation steps is shown in figure 3.3. This interactive program requires one final step to become a CGI program. The revision in figure 3.4 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 3.2.

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

```
(prompt-read-k "Enter ... first ... "
  (lambda (res1)
    (prompt-read-k "Enter ... second ...."
      (lambda (res2)
        (display (+ res1 res2)))))))
```

where

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

In other words, the CPS converter must supply alternate implementations of primitives. For higher-order primitives, such as *map*, it must add CPS'ed versions so that the call-backs are applied to a continuation, which may after all represent resumption points. External modules that accept function arguments must be transformed as well. We use the CPS conversion of Danvy and Filinski to avoid introducing administrative beta redexes [12, 43].

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

```
'(lambda ,args ,body ...)
```

is replaced with

```
'((lambda ,fv (lambda ,args ,body ...)) ,fv)
```

where *fv* is the list of free variables in *body* This new function is closed, so it can be safely lifted to the outermost lexical scope.

For our running example, this step yields

```
(define closure1
  (lambda ()
    (lambda (res1)
      (prompt-read-k "Enter ... second ...."
        [closure2 res1])))))
```



```

(define closure2
  (lambda (res1)
    (lambda (res2)
      (display (+ res1 res2))))))

(prompt-read-k "Enter ... first ...." [closure1])

```

Using *closure1* and *closure2* we can now run the program from different resumption points, turning the original program into a curried adder just as the back button on a Web browser does.

Figure 3.3 shows the result of the final compilation step, namely of converting closures into **structs**; 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 (sub) program to invoke next. The continuation's environment captures any values needed by the next sub program instead of explicitly passing them in hidden fields.

Up to this point, the transformation produced a semantically equivalent program, so that the result is a normal interactive program. To produce a CGI program, we replace two pieces of the lambda-lifted closure-converted program. The definition of *prompt-read* changes and now marshals the continuation into a Web form, asks the supplied question, and then exits. The expression changes to the text of figure 3.4. 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.

3.5 Compiling Stateful CGI Programs

```
(define box-0 (box 0))
(define box-1 (box 0))

(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 3.5 : A Stateful Interactive Program

While generating CGI programs from interactive functional programs is almost a routine task with functional compilation techniques, *internal*² mutations in the interactive program pose an interesting challenge. The first problem is due to plain variable assignments—`set!` in Scheme—because lambda lifting and closure conversion assume that copying bindings is acceptable. We must therefore eliminate all assignment statements with a transformation that replaces mutable variables by cells (boxes in Scheme), assignments to variables with assignments to cells, and references to such variables with dereferences of cells. Furthermore, the CGI program generator must know all cells that the original program uses (or implicitly introduces).

The second problem is much more severe. Semantically, mutations 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 is different from the store of the first one. Modifications of the store survive continuation capture and invocation.

A consumer who invokes the same continuation twice via a Web form should also see that the store modifications of the first invocation survive till the second invocation

²We ignore mutations of *external* entities, say the server file system or a database, because this topic is well-understood.

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 with the form, the browser would also copy the store, and two submissions of the form would submit the same store twice.

Still, we must choose where to remember the current store when we suspend a CGI program. We could either place the store on the server or on the client machine. As we already know from the discussion of the placement of continuations, the server is ill-suited for this purpose. Hence, we must turn the store into a datum that is sent to, and then stored on, the consumer's machine—but not inside the Web page.

This reasoning leaves us with the single choice of turning the store into a browser “cookie” and placing this marshaled form into the consumer's cookie file. Unlike hidden fields, they are independent from any particular page, so changing continuations via the back button does not affect the store.

Although this naïve cookie solution sounds straightforward, it has two imperfections. The first one, which is minor, is the restriction that Web browsers have a limit of 80kB of storage for cookies per host name [36]. 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 may be problematic for some programs. As security research improves, we expect these simplistic limitations to be lifted. The second, more important, one arises because browsers transmit cookies at the time they submit the Web request. If the user submits simultaneous requests, one of the cookie transmissions may contain an out-of-date store. A naïve implementation may thus lose updates to the store.

Our solution is to include a sequence number [41] with the cookie store. A sequence number 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, though 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: continuations with environments and stores. Our CGI compiler can therefore use the browsers' mechanisms to implement the separate storage requirements for continuations and stores in a systematic manner. In this context, it is interesting to note that the size limitations for cookies almost forces CGI programmers to design their programs in a mostly functional style, whether or not they use functional languages.

3.6 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 debugging 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 such as

```
Premature end of script headers
```

followed by the name of the program. From this, the programmer can infer that the CGI program did not output a valid response before terminating, but little more about the error.

Our 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.

We can overcome both problems with a minor modification of existing programming environments. 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 the 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.

To test this idea, we wrote a small library (technically, a Teachpack [15]) of interaction functions for DrScheme, our programming environment [15] for Scheme. The re-implemented *prompt-read* primitive accepts HTML pages (with forms); it grabs the current continuation, stores it, and manages the communication with the browser.

Without further ado, 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 S-expressions, and DrScheme highlights the place where the program raised the exception as if the program were an ordinary interactive program. See figure 3.7 for an illustration.

Consider the more complex example of DrScheme's stepper tool [9]. The tool reduces Scheme programs according to Scheme's reduction semantics [14]. 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 3.8 for an illustration of this capability.

In general, our methodology for developing CGI programs permits the use of conventional design methods for interactive programs and the use of systematically enriched programming environments. We believe that our ideas thus bring rigorous order to the world of CGI programming.

3.7 Implementation Status

Both the CGI compiler and the CGI Teachpack for DrScheme exist in prototype form. We have developed a number of examples in this context, plus one full-fledged application: the teacher enrollment dialog for our TeachScheme! outreach project.

Our prototype CGI compiler operates on R4RS [10] programs without global **defines** and without multiple values. The compiler accepts a single expression, typically a **letrec** expression prefixed with PLT Scheme library specifications. Currently the compiler accepts only **letrecs** with values on the right-hand side.

The marshaling primitives use the standard Scheme printer, which automatically takes care of sharing and cycles in the reading and writing of environments and cookies. Higher-order data, including continuations and closures, marshal correctly since closure conversion represents them as vectors. The encoding could benefit from a compression step [29] to reduce network traffic and the amount of data that is stored in cookies.

```

(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 3.3
(define closures (vector ...))

;; 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-compatible HTTP header and HTML Web form
;; including a cookie containing the-boxes
(define (produce-html question mark free-values)
  ... (write-boxes-to-cookie the-boxes)...)

(define bindings (get-bindings))

;; the-boxes : (vectorof Value), the current store
(define the-boxes
  (if (null? bindings)
    (initialize-the-boxes)
    (read-boxes-from-cookie)))

;; initialize-the-boxes : → (vectorof Value)
;; create a new store plus a sequence number
;; read-boxes-from-cookie : → (vectorof Value)
;; turn a cookie into a store, check sequence number using a lock file
;; write-boxes-to-cookie : (vectorof Value) → void
;; turn a store into a cookie, increment sequence number using a lock file

(cond
  [(null? bindings)
   (apply-closure (make-closure 0 '()) (box 0))]
  [else
   (apply-closure (make-closure (string→number
                                (extract-bindings/single 'cont bindings))
                                (create-env-from-strings
                                 (extract-bindings/single 'env bindings)))
                  (extract-binding/single 'response bindings))])

```

Figure 3.6 : A Stateful CGI Version

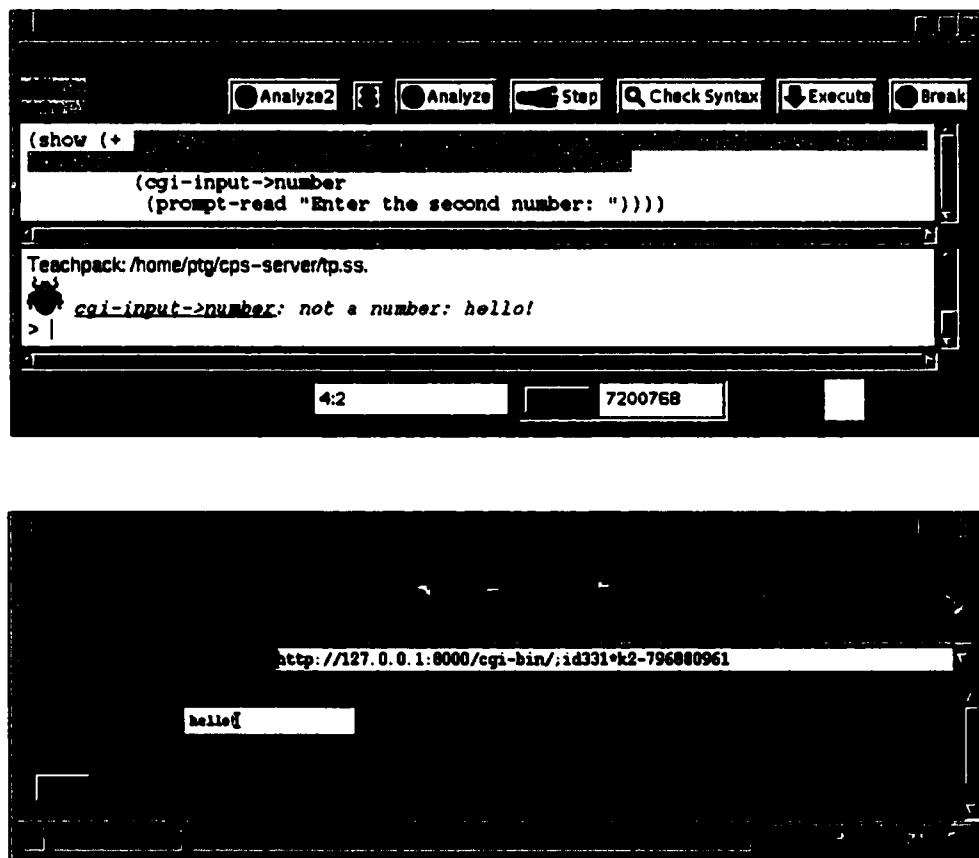


Figure 3.7 : CGI Error Reporting

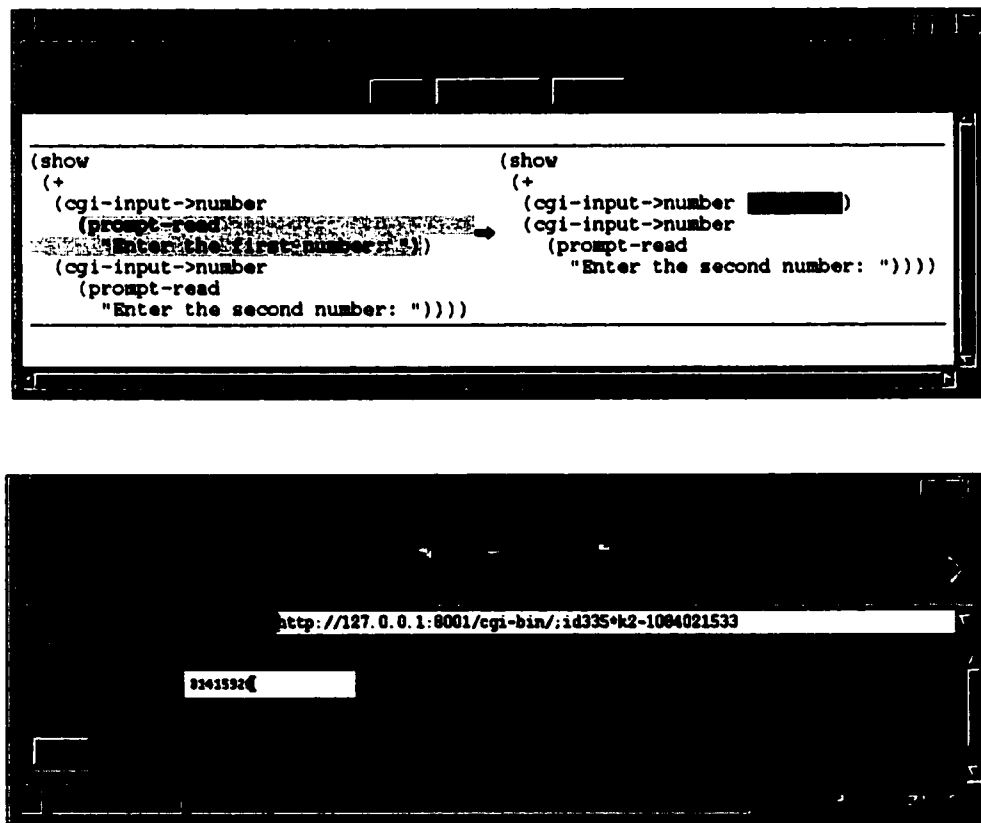


Figure 3.8 : CGI Stepping

Chapter 4

Related Work

Others laid the ground work for developing interactive CGI programs. Hughes [27] developed arrows as a generalization of monads. One of his example sections describes how to implement interactive CGI programs using arrows. His key insight is to provide a mechanism that turns the current continuation into a datum for the Web page. Similarly, Queinnec [40] advocates using *call/cc* to implement interactions between Web servers and consumers. His method requires extending the server to store continuations.

Each approach suffers from a shortcoming: the arrow solution deals with stores improperly, while storing continuations in a Web server induces time-outs. Our compilation based solution overcomes both these difficulties. Furthermore, our work demonstrates that these ideas are applicable to all kinds of languages, not only languages that support monads and arrows or functional languages with *call/cc*.

The performance problems of the CGI interface has led others to develop higher-speed alternatives [37, 46]. In fact, one of the driving motivations behind the Microsoft .NET initiative [32] appears to be the need to improve Web server performance by eliminating process boundaries.¹

Apache allows programmers to eliminate process boundaries by providing an interface to modules [46] extending the server with code that, among other things, generates content dynamically for given URLs. However, circumventing the underlying operating system without providing an alternative protection mechanism within the server opens the entire server to catastrophic failures.

¹Shriram Krishnamurthi relayed his personal communication with Jim Miller to me.

FastCGI [37] provides a safer alternative by placing each content generator in its own process. Unlike traditional CGI, FastCGI processes handle multiple requests, thereby avoiding the process creation overhead.

At first glance, a reader could suspect that the FastCGI protocol solves the problems of CGI program design and implementation. Since these programs specifically wait for a request, it may appear 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. However, this approach 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 in the program.

Furthermore, FastCGI's use of a separate process generates bi-directional inter-process communication cost and introduces coherence problems in the presence of state. It also complicates the creation of cooperative CGI programs that communicate higher-order data.

The **<bigwig>** system [7] also recognizes the need for designing interactive systems. However, they claim that users should not be allowed to go back to previously visited pages since programs don't handle this well. Thus, similar to FastCGI, their system delegates requests to persistent session threads that suspend at each interaction point, as mentioned above. Instead of responding directly to requests, the system writes the response to a file and redirects the client to that document. This purposely disables the use of bookmarks or the back button to resume prior points in the computation since bookmarks always return to the most recent interaction and the back button starts the session over. We believe that people change their minds and do want to return to prior interactions, so Web applications must handle this situation rather than disallowing it.

The performance of **<bigwig>** programs does not look promising, though this is acceptable for applications with few users. Handling a request involves creating a

new “connector” process, thus introducing the overhead of the CGI protocol. It also involves the extra interprocess communication overhead of FastCGI. Furthermore, redirecting each CGI request to a file introduces extra network overhead, not to mention the cost of reading and writing files.

On a positive note, **<bigwig>** provides sophisticated handling of concurrency using Monadic Second-order Logic on Strings. Our server serializes requests for a given session. For collaborative CGI programs our server provides the programmer with more mundane concurrency control operations based on semaphores. Their system also compiles regular expression restrictions on HTML form fields into JavaScript that checks the form values before submission. The server rechecks the values in case the client bypasses the checks. Our system relies solely on server side checking.

IO-Lite [38] demonstrates the performance advantages of programs modified to share immutable buffers instead of copying mutable buffers across protection domains. Since the underlying memory model of MrEd provides safety and immutable string buffers, our server automatically provides this memory model without the need to alter programming styles or APIs.

The problem of managing resources in long-running applications has been identified before in the Apache module system [46], in work on resource containers [6], 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.

Shivers also implemented an extensible Web server in Scheme [44]. Instead of exporting a function that handles each request for a URL from a replaceable unit, his server accepts this function as an argument. His system also provides a library of default request handling functions and a means of combining them. Although the system provides a means of uploading code to the server and running it in a controlled environment, it does not address resource management issues for CGI

programs. He may not have addressed this problem because his server does not directly support interactive programs via continuations, which increases the need for resource management.

The FoxNet project [25] implemented a modular Web server in the high level language SML [33], and demonstrated that it performed well. This idea of integrating languages and systems dates back as far as Lisp machines [42], which implemented Lisp directly with special purpose hardware.

Like our server programs, Java servlets [11] are content-generating code that runs in the same runtime system as the server. Without the subprocess management facilities of MrEd's custodians, the server relies 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 is not guaranteed to comply.

Aside from the object-oriented interface and libraries for constructing HTTP response headers, servlets provide the same programming model as standard CGI. Each incoming request invokes a method in the servlet, leaving the task of restoring the appropriate control context to the programmer. It may appear that servlets can avoid moving the store into cookies by holding values in the servlet object's fields. However, the Web server has the option of unloading a servlet and creating a new one at any time. 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. The `HttpSession` class provides a mechanism for maintaining a dictionary from Strings to Objects on the server and storing a reference to the dictionary in a URL, cookie, or Secure Sockets Layer session. All the problems with server side state consuming memory and timing out remain.

The J-Server [45] runs atop Java extended with operating systems features. The J-Server team identified and addressed the server's need to prevent dynamic content generators from shutting down the entire server, while allowing the server to shut-down the content generators reliably. They too identified the need for generators to

communicate with each other, but their solution employs remote method invocation (which introduces both cost and coherence concerns) instead of shared 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. Thus, their programming model is the same as CGI, FastCGI, and Java servlets.

Chapter 5

Future Work

Two major areas of future work involve type systems and interoperability. Research should explore how the essential additions to Scheme—dynamically linkable modules that are first-class values, threads, custodians, and parameters—can be integrated in typed functional languages such as ML and how the type system can be exploited to provide even more safety guarantees. While MrEd already permits programmers to interoperate with C programs through a foreign-function interface, we are studying the addition of and interoperation between multiple safe languages in our operating system, so programmers can use the language of their choice and reuse existing applications [31].

The compilation based implementation does not address security. Malicious users could examine the hidden field containing the marshaled continuation, possibly revealing confidential information the programmer intended to remain internal to the program. Worse, the user can modify the continuation including any of its free variables. To prevent these problems, the system needs to represent the continuation in an illegible, unforgeable manner. Some form of encryption or digital signature may suffice.

While storing information in the client introduces security concerns, the custom server's strategy of keeping the data in the server's memory loses data if the server goes down. This inconveniences the programmer with the task of explicitly storing more important data on disk or in an external database. Instead, the system should checkpoint the program's state periodically or at each interaction, so maintaining values in ordinary program variables would become robust.

The superior performance of the custom server for dynamically generated content should substantially outweigh performance penalties of added security or checkpointing. However, the performance of the server for static files could be improved further by various caching techniques and by adding persistent connections. Further measurements could directly compare our system's dynamic performance to other systems instead of indirectly comparing the other systems to CGI or to FastCGI.

In addition to relaxing the memory performance, *send/finish* allows the programmer to constrain the program's control flow. Although adequate for tasks with one final, irreversible action at the end of the program, it lacks the flexibility required for other tasks. For example, the programmer may want to prevent the user from going back prior to a given point, but still allow forward progress, similar to Prolog's cut operation. Many constructs such as *dynamic-wind* [26], *F* and *prompt* [43], *shift* and *reset* [12], and others exist. Which operations restrict Web computations in desirable ways remains an interesting question.

Chapter 6

Conclusions

The content of the Web is becoming more dynamic. The next generation of Web servers must therefore tightly integrate and support the construction of extensible and verifiable dynamic content generators. Furthermore, they must allow programmers to write interactive, dynamic scripts in a more natural fashion than engendered by current Web scripting facilities. Finally, the servers must themselves become more extensible and customizable.

This dissertation demonstrates that all these programming problems can be solved with a high-level programming language, provided it offers OS-style services in a safe manner. Our server accommodates both kinds of extensibility found in traditional servers—applications, which serve data, and extensions, which adapt the behavior of the server itself—by exploiting its underlying module system. All these features are available on the wide variety of platforms that run MrEd (both traditional operating systems and experimental kernels such as the OS/Kit [19]). The result is a well-performing Web server that accommodates natural and flexible programming paradigms without burdening programmers with platform-specific facilities or complex, error-prone dynamic protocols.

This dissertation also demonstrates why compilation techniques from the functional world matter for Web programming. It shows that one can design interactive programs and then translate them into CGI compliant structure using CPS conversion, box conversion, lambda lifting and closure conversion, followed by the generation of a little administrative stub code. Furthermore, bringing this systematic order to the world of CGI programming also solves the problem of developing CGI programs

in a conventional programming environment.

Our work applies to conventional languages as well as functional languages. We can clearly implement these transformations for languages such as Perl [48], Python [47] and Java [22]. Indeed, since Python now supports a form of continuation operator, we can also turn IDLE [13] into a CGI development environment.

Even in the absence of such tools, we can use the CGI example to motivate the teaching of CPS conversion, lambda lifting, and so on in our programming languages and compilers courses. The dissertation shows that learning such transformation techniques is not just an idle exercise in functional programming but a valuable skill in the world of Web programming.

The dynamic, *call/cc* based technique and the static, compilation based technique each have advantages and disadvantages. The dynamic approach requires a custom Web server, while the compilation approach produces standard CGI programs that run on any server. Integrating the Web applications into the server dramatically improves performance and eliminates the need for an underlying operating system. Storing continuations on the server increases the server's memory usage and requires timeouts. Marshalling them onto the client introduces extra network traffic and security concerns. Keeping a separate store on each client also precludes cooperation among Web applications through shared lexical variables.

For non-collaborating Web applications, the best solution for constructing CGI programs combines the two techniques. The compiler eliminates calls to *send/suspend* statically, leaving the server free from storing continuations. The custom server then dynamically extends itself with the resulting program, so the code runs quickly.

Bibliography

- [1] Acme Labs. Web server comparisons.
<http://www.acme.com/software/thttpd/benchmarks.html>.
- [2] Apache. <http://www.apache.org/>.
- [3] Arlitt, M. and C. Williamson. Web server workload characterization: the search for invariants. In *ACM SIGMETRICS*, 1996.
- [4] 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.
- [5] Banga, G. and P. Druschel. Measuring the capacity of a web server. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [6] 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.
- [7] Brabrand, C., A. Møller, A. Sandholm and M. I. Schwartzbach. A runtime system for interactive web services. In *Journal of Computer Networks*, 1999.
- [8] BrightPlanet. DeepWeb.
<http://www.completeplanet.com/Tutorials/DeepWeb/>.
- [9] Clements, J., M. Flatt and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, 2001.

- [10] Clinger, W. and J. Rees. Revised⁴ report on the algorithmic language Scheme. In *ACM Lisp Pointers*, pages 1–55, 1991.
- [11] Coward, D. Java servlet specification version 2.3, October 2000.
<http://java.sun.com/products/servlet/index.html>.
- [12] Danvy, O. and A. Filinski. Representing control: a study of the CPS transformation. In *Mathematical Structures in Computer Science*, volume 2, pages 361–391, 1992.
- [13] Daryl Harms. Using IDLE. <http://www.python.org/idle/doc/>.
- [14] 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.
- [15] 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. to appear.
- [16] 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.
- [17] Flatt, M. and M. Felleisen. Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998.
- [18] 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.

- [19] Ford, B., G. Back, G. Benson, J. Lepreau, A. Lin and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *16th ACM Symposium on Operating Systems Principles*, October 1997. Saint-Malo, France.
- [20] Friedman, D. P., M. Wand and C. T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, 1992.
- [21] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [22] Gosling, J., B. Joy and G. L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.
- [23] Graunke, P., R. B. Findler, S. Krishnamurthi and M. Felleisen. How to design and generate cgi programs: Applying functional compiler techniques to the real world. In *ACM SIGPLAN International Conference on Functional Programming*, September 2001. Submitted for publication.
- [24] Graunke, P., S. Krishnamurthi, S. van der Hoeven and M. Felleisen. Programming the web with high-level programming languages. In *European Symposium on Programming*, 2001.
- [25] Harper, R. and P. Lee. Advanced languages for systems software: The Fox project in 1994. Technical Report CMU-CS-94-104, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1994. (Also published as Fox Memorandum CMU-CS-FOX-94-01).
- [26] Haynes, C. T. and D. P. Friedman. Constraining control. In *12th ACM Symposium on Principles of Programming Languages*, 1985.
- [27] Hughes, J. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67-111, May 2000.
- [28] Jackson, M. A. *Principles of Program Design*. Academic Press, 1975.

- [29] Jansson, P. and J. Jeuring. Polytypic compact printing and parsing. In Swierstra, S. D., editor, *ESOP: Proceedings European Symposium on Programming*, pages 273–287. Springer-Verlag, 1999. LNCS 1576.
- [30] Kelsey, R., W. Clinger and J. Rees. Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.
- [31] Krishnamurthi, S. *Linguistic Reuse*. PhD thesis, Rice University, 2001.
- [32] Microsoft Corporation. <http://www.microsoft.com/net/>.
- [33] Milner, R., M. Tofte, R. Harper and D. MacQueen. The denition of Standard ML (revised), 1997.
- [34] Mogul, J. The case for persistent connection HTTP. In *ACM SIGCOMM*, 1995.
- [35] NCSA. The common gateway interface. <http://hoohoo.ncsa.uiuc.edu/cgi/>.
- [36] Netscape Communications Corporation.
http://www.netscape.com/newsref/std/cookie_spec.html.
- [37] Open Market, Inc. FastCGI specification. <http://www.fastcgi.com/>.
- [38] 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.
- [39] Pitman, K. Special forms in Lisp. In *Conference Record of the Lisp Conference*, August 1980. Stanford University.
- [40] Queinnec, C. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.

- [41] Reed, D. P. Implementing atomic actions on decentralized data. In *ACM Transactions on Computer Systems*, pages 234–254, February 1983.
- [42] Richard, G. and L. Machine. Mit artificial intelligence laboratory, 1974.
- [43] Sabry, A. and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 1993.
- [44] Shivers, O. The Scheme Underground web system.
<http://www.swiss.ai.mit.edu/ftplib/scsh/contrib/net/su-httpd.html>.
- [45] 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.
- [46] Thau, R. Design considerations for the Apache server API. In *Fifth International World Wide Web Conference*, May 1996.
- [47] van Rossum, G. Python reference manual. Technical report, Corporation for National Research Initiatives (CNRI), October 1996.
- [48] Wall, L. and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1992.