Automatically Restructuring Programs for the Web

Paul Graunke
Northeastern University
ptg@acm.org

Robert Bruce Findler
Rice University
robby@cs.rice.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract

The construction of interactive server-side Web applications differs substantially from the construction of traditional interactive programs. In contrast, existing Web programming paradigms force programmers to save and restore control state between user interactions. We present an automated transformation that converts traditional interactive programs into standard CGI programs. This enables reuse of existing software development methodologies. Furthermore, an adaptation of existing programming environments supports the development of Web programs.

1 Designing Web Programs

The need for generating Web information on demand is obvious. One 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, Common Gateway Interface (CGI) programs halt after processing a single form. Similarly, Java servlet 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 paper's discussion 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.

In this paper, we show that Web programmers can use *existing* software engineering methods to develop interactive programs and that well-known, automatable transfor-

mations can *generate* standard CGI scripts from these programs. Specifically, we extend a programming language with a primitive for Web interactions and show how this extension 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 we can adapt existing programming environments in support of this development style.

The remainder of this paper is organized as follows. The second section of this paper is a brief introduction to conventional Web programming. The third section presents the central ideas of the paper: the new I/O construct and its implementation. 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 how we have implemented our ideas in Scheme [7], so that we can test each development stage. The work does not rely on Scheme's advanced control constructs, however. The sixth section discusses related work. We discuss in the seventh and last section how our ideas carry over to many other programming languages, even those without Scheme's advanced control constructs, and how they are useful even in the absence of tool support.

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 [13] 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 the Web, first via conventional means and then in a more direct manner.

2.1 Conventional CGI Programs

Figure 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 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 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 inverted[1]—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

[1] M. Jackson [22] 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 programs that don't support such forms of control.

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

;; main
(display                                                          3
  (+ (prompt-read "Enter the first number to add:")               1
     (prompt-read "Enter the second number to add:")              2))
```

Figure 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
(cond   ;; 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 '())                    1]
  [(string=? (extract-binding/single 'continue-at bindings) FIRST-STOP)  ;; 'continue-at is a symbol, a string optimized for equality comparison
   (produce-html "Enter the second number to add: " SECOND-STOP
                 (list (list 'first-number (extract-binding/single 'response bindings))))   2]
  [(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))))          3])
```

Figure 2. A CGI Version of Figure 1

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. One of the authors recently renewed 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.

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, we propose that 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 pro-

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

gram, with no intervention from the programmer

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 reimplementations of primitives such as display or prompt-read. A specialized version of prompt-read can capture the current control state as a continuation [33] 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.

Prior work [19, 28] 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 severe 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 don't 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 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 contains 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.

3 Generating CGI Programs

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

3.1 Functional CGI Programs

Removing timeouts would eliminate many of the problems encountered with our 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. To eliminate the uses of *call/cc* to suspend continuations, we utilize techniques for compiling functional programming languages. More specifically, we employ three well-known transformations to automatically create the control flow required for Web applications:

Continuation Passing Style (CPS) [15] 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 our case, a reimplementation 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 [24] 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 [30] changes the representation of

[3] Also, 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
    (lambda ()                ;; the environment (in this case, empty)
      (lambda (response1)  ;; the argument
        (prompt-read-k "Enter the second number to add:" (make-closure 1 (list response1)))))

     (lambda (response1)      ;; the environment (in this case, holds the previous argument)
       (lambda (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 3. The Compiled Version of Figure 1

higher-order data, such as closures[4] and continuations, into a first-order form. By choosing portable concrete representations (in this case, vectors), we 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.

We explain the process with the trivial but illustrative example from figure 1. The result of these three automated translation steps is shown in figure 3. This interactive program requires one final step to become a CGI program. The revision in figure 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 2.

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

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 declares anonymous, first-class functions
  (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)))
```

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

```
(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

(define closures ...) ; as in figure 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. The CGI Version of Figure 3 (Compare with Figure 2)

to resume a continuation with a call to a global function. Each expression of the form

(**lambda** ⟨*args*⟩ ⟨*body*⟩ ...)

is replaced with

((**lambda** ⟨*free-vars*⟩
  (**lambda** ⟨*args*⟩ ⟨*body*⟩ ...))
⟨*free-vars*⟩)

where ⟨*free-vars*⟩ 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 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 hid-

den fields.

Up to this point, the transformation produced a semantically equivalent program, so the result is a normal interactive program. To produce a CGI program, we replace two fragments of the defunctionalized 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. 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.

To prove the well-behaved nature of our transformation, we would need to construct 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. We intend to investigate a formal proof along these lines in future work.

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 [3] 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 [9] simultaneously does block encryption as well as message authentication in one (highly parallelizable) operation.

3.2 Compiling Stateful CGI Programs

While generating CGI programs from interactive functional programs is almost a routine task with functional compi-

```
(define box-0 (box 0))
(define 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 5. A Stateful Interactive Program

```
(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
(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 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 (empty-bindings? 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

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

Figure 6. Its CGI Version

lation 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. We must therefore eliminate all assignment statements 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 5 contains an imperative version of our 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.

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, 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.[7] 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 marshalled 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. Figure 6 sketches the cookie-based translation of figure 5.

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 [26]. 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, we expect cookies or some other mechanism to mature enough to lift these simplistic restrictions. The second, more important, one 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 out-of-date cookie. A naïve implementation may thus lose updates to the store.

Our solution is to include a sequence number [29] 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 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. Our CGI compiler can therefore use the browsers' mechanisms to implement the separate storage requirements for continuations and stores in a systematic manner.

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.

Our compilation process introduces the additional problem that the code that is executed as a CGI script is not the

[5] We ignore modifications of data in *external* entities, say the server file system or a database, because this topic is well-understood.

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

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

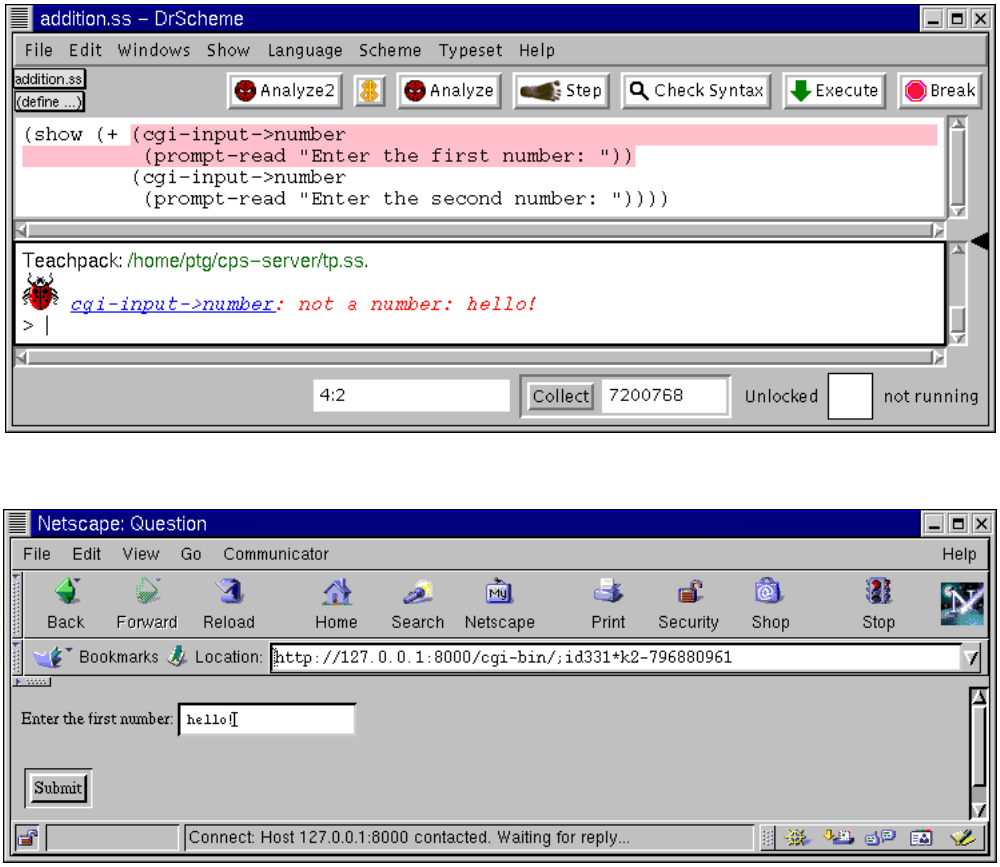

Figure 7. CGI Error Reporting

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 demonstrate this idea, we wrote a library (technically, a Teachpack [14]) of interaction functions for DrScheme, our programming environment [14] for Scheme. 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 7 for an illustration.

Consider the more complex example of DrScheme's single-step debugger [6]. The tool reduces Scheme programs according to Scheme's reduction semantics [12]. 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

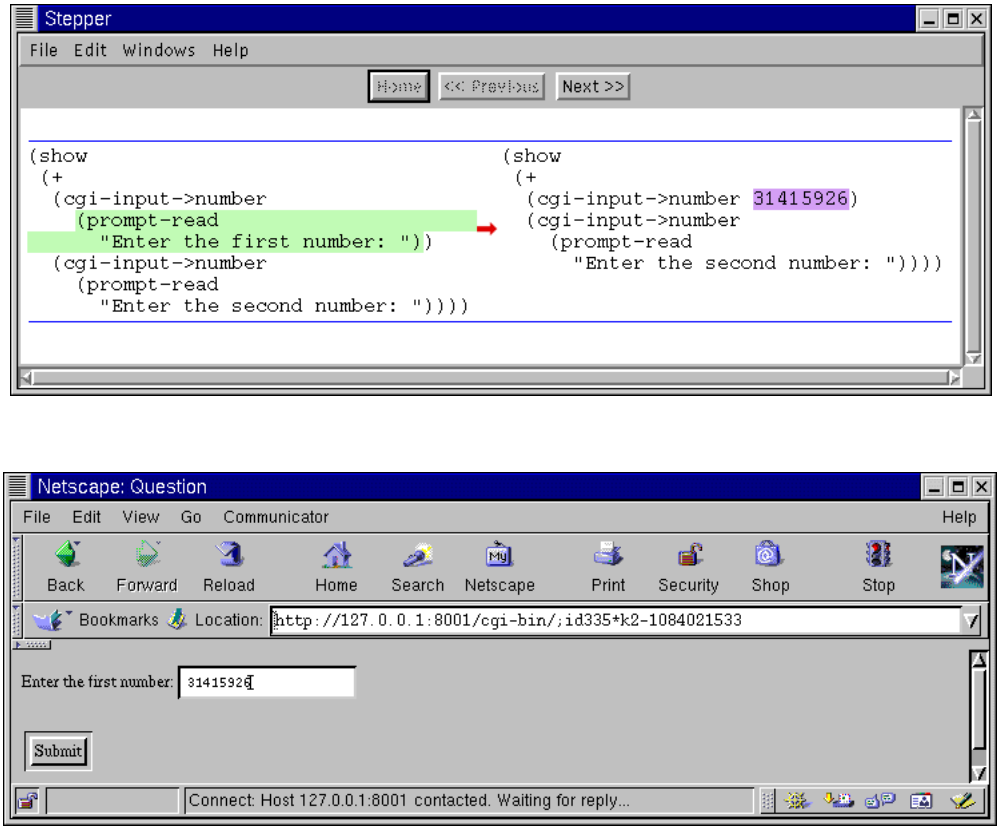

Figure 8. CGI Stepping

output steps. See figure 8 for an illustration of this capability.

In general, our methodology for developing CGI programs permits the use of conventional software engineering 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.

5 Implementation Status

Both the CGI compiler and the CGI Teachpack for DrScheme exist in prototype form. Our prototype CGI compiler operates on R4RS [7] programs with some minor restrictions. We have developed a number of examples in this context, plus one full-fledged application: the teacher enrollment dialog for an outreach project.

The marshaling primitives use the existing PLT Scheme printer, 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 [23] to reduce network traffic and the amount of data that is stored in cookies. We use the CPS conversion of Danvy and Filinski to avoid introducing administrative beta redexes [10, 31].

6 Related Work

Programmers building imperative-style programs in purely functional languages use a technique based on the mathematical theory of monads. Hughes [20], 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. Similarly, Queinnec [28] advocates using *call/cc* to implement interactions between Web servers and consumers. His method requires the modification of a server that can store continuations.

Our research started as an exploration of these two publications. We diagnosed the short-comings of these approaches, namely, that the arrow solution deals with stores

improperly and the time-outs, based on our experience, dog continuation objects in a Web server. Our solution addresses both problems and overcomes these difficulties. Furthermore, our work demonstrates that these ideas are applicable to all kinds of languages, not only functional languages supporting first-class continuations.

Graham [18] 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 manually, using our automated translation must be even better. He does not explain how his company dealt with mutable stores.

At first glance, a reader might suspect that the FastCGI protocol [27] 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. 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, causing the FastCGI program to either not find fields expected from the correct form or, even worse, to misinterpret fields that accidentally coincide.

The Mawl system [4] 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.

Java servlets [8] address performance issues in a manner similar to FastCGI. 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 `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 store into cookies by storing values in the servlet object's fields. However, the Web server has the option of garbage collecting 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 or timing out remain.

The Java Platform Debugger Architecture [34] enables Java development environments [5, 21, 35] to attach remotely to the JVM that the Web server uses to run 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.

7 Conclusion

Our paper introduces an automated translation that implements an interactive programming model for Web applications. This model matches the mental model of software engineers accustomed to thinking about traditional interactive programs. By avoiding the manual saving and restoring of control state between interactions, the system not only eases the initial software development, but also facilitates maintenance and assists other engineers in understanding the product. Software engineers can port legacy software to the Web by using our transformation. Furthermore, bringing this systematic order to the world of CGI programming solves the problem of developing CGI programs. Futhermore, our technique allows developers to use conventional programming environments.

The automated translation produces CGI-compliant programs using CPS conversion, box conversion, lambda lifting and defunctionalization, followed by the generation of a little administrative stub code. The well-understood formal nature of the first four steps justifies a high degree of confidence in the translation process.

We can implement these transformations for languages such as Perl [38], Python [37] and Java [17]. It is easy to simulate closures with objects, but the lack of tail-call optimization [25] makes it difficult to control stack growth. We could use exceptions [16, 32] to ameliorate this problem. Indeed, since Python now supports a form of continuation operator [36], we can also turn IDLE [11] into a CGI development environment. Even in the absence of such tools, programmers can achieve a lesser degree of benefit by proceeding in a systematic manner and documenting the design pattern.

Acknowledgements

We thank Morgan McGuire and the anonymous referees for their comments.

References

[1] `http://www.htdp.org/`.

[2] http://www.teach-scheme.org/.

[3] Keyed-hash message authentication code (HMAC). http://www.nist.gov/hmac.

[4] D. L. Atkins, T. Ball, G. Bruns, and K. C. Cox. Mawl: A domain-specific language for form-based services. *Software Engineering*, 25(3):334–346, 1999.

[5] Borland Software Corporation. http://www.borland.com/jbuilder/.

[6] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, 2001.

[7] W. Clinger and J. Rees. Revised^4 report on the algorithmic language scheme. In *ACM Lisp Pointers*, pages 1–55, 1991.

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

[9] J. Daemen and V. Rijmen. Advanced Encryption Standard. http://www.nist.gov/aes.

[10] O. Danvy and A. Filinski. Representing control: a study of the CPS transformation. In *Mathematical Structures in Computer Science*, volume 2, pages 361–391, 1992.

[11] Daryl Harms. Using IDLE. http://www.python.org/idle/doc/.

[12] M. Felleisen 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.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, June 1999. Internet Request for Comments 2616.

[14] R. B. Findler, 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.

[15] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, 1992.

[16] S. Funfrocken. 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 1998. LNCS 1477.

[17] J. Gosling, B. Joy, and G. L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[18] P. Graham. Lisp in web-based applications. http://www.paulgraham.com/articles.html.

[19] P. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the web with high-level programming languages. In *European Symposium on Programming*, 2001.

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

[21] International Business Machines, Inc. http://www.ibm.com/websphere.

[22] M. A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[23] P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In S. D. Swierstra, editor, *ESOP: Proceedings European Symposium on Programming*, pages 273–287. Springer-Verlag, 1999. LNCS 1576.

[24] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985.

[25] G. L. S. Jr. Debunking the 'expensive procedure call' myth. In *Proceedings of the ACM National Conference*, pages 133–162, October 1977.

[26] Netscape Communications Corporation. http://www.netscape.com/newsref/std/cookie_spec.html.

[27] Open Market, Inc. FastCGI specification. http://www.fastcgi.com/.

[28] C. Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.

[29] D. P. Reed. Implementing atomic actions on decentralized data. In *ACM Transactions on Computer Systems*, pages 234–254, February 1983.

[30] J. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, 1972.

[31] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 1993.

[32] T. Sakamoto, 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 (ASA/MA)*, pages 16–28, September 2000.

[33] C. Strachey 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.

[34] Sun Microsystems, Inc. http://java.sun.com/products/jpda/.

[35] Sun Microsystems, Inc. Forte tools. http://www.sun.com/forte/.

[36] C. Tismer. Continuations and stackless python... In *Proceedings of the 8th International Python Conference*, January 2000.

[37] G. van Rossum. Python reference manual. Technical report, Corporation for National Research Initiatives (CNRI), October 1996.

[38] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1992.