Automatically Restructuring Programs for the Web[†]

Jacob Matthews (jacobm@cs.uchicago.edu)
University of Chicago

Robert Bruce Findler (robby@cs.uchicago.edu)
University of Chicago

Paul Graunke (ptg@acm.org)
Fleet Numerical Meteorology and Oceanography Center (FNMOC)

Shriram Krishnamurthi (sk@cs.brown.edu)
Brown University

Matthias Felleisen (matthias@ccs.neu.edu)
Northeastern University

[†] This work was supported by several grants from the National Science Foundation.

1. Designing Web Programs

Nowadays, many Web pages are generated on demand. 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) [33] programs halt after processing a single form. Similarly, Java servlets [12] and Java Server Pages [39] must respond with a single page and then terminate, in response to a single interaction with the user. That is, all widely used Web technologies suffer from the same problem: the program's control information is erased between interactions with the user.

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 common Web programming languages, the designer cannot match the structure of the interaction with the structure of the program. Indeed, maintaining control information properly is a general problem when the interface to a program is a Web browser. It results in programmers setting up ad hoc mechanisms to save and restore control state that are difficult to develop, maintain, or explain to colleagues. Even session-management features built in to various languages designed for generating Web pages only solve the easier half of the problem, automatically saving variable bindings but not control state.

In this paper, we show that Web programmers can use existing software engineering methods to develop interactive programs and that well-known, algorithmic transformations 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 construct, its implementation for a minimal core Scheme [31] dialect, and how to use the same implementation technique in languages without Scheme's advanced control constructs. The fourth section illustrates that when Web programs are just interactive programs, programmers can develop, test, and debug them in ordinary programming environments, enriched with a small run-time extension. The fifth section outlines how we have implemented our ideas in PLT Scheme [19], so that we can test each development stage. The sixth section discusses related work. In the seventh and last section we conclude and discuss a few areas of future research.

2. Interactive CGI Programs

```
;; 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)) ;; the name=value pairs submitted on this invocation;
                                 ;; access is via extract-bindings/single
;; main
(cond   ;; each bracketed clause is a question-answer pair;
        ;; in this instance, all answer expressions are boxed.
  [(empty-bindings? bindings) ; user submitted no 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. Scheme CGI Versions of Figure 1

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 [15] 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.

For concreteness, we focus on CGI script programming for the rest of this paper, but these techniques apply to any Web technology that erases the control context between interactions with the user.

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 exist only 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

```
<?
function extract_binding($name) { ...}
function produce_html($question, $mark, $freevars) { ?> ...<? }

function display($ans) { ?>
  <HTML>
     <BODY>The answer is <?= $ans ?>.</BODY>
  </HTML> <?}

$FIRST_STOP = "first number done";
$SECOND_STOP = "second number done";

$stop = extract_binding("continue_at");
if (!isset($stop)) {
    produce_html("Enter the first number to add:", $FIRST_STOP, array());
} else if ($stop === $FIRST_STOP) {
  produce_html("Enter the second number to add:",
               $SECOND_STOP,
               array("first_number" => extract_binding("response")));
} else if ($stop === $SECOND_STOP) {
  display(extract_binding("first_number") + extract_binding("response"));
}
?>
```

Figure 3. PHP CGI Versions of Figure 1

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. This is complicated by the consumer's ability to use the back button, clone the window, or bookmark and return multiple times, thus becoming not just a second coroutine but a third, fourth, fifth, or any arbitrary number of coroutines all interacting with the same script. 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 'continue-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 'continue-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.

Even PHP [7], a programming language designed for web scripting, suffers from the same problems. Figure 3 shows the same adder program, this time written in PHP.

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 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. For example, one of the authors renewed two Internet domain name registrations. The

[1] M. Jackson 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* [29], a technique for providing simple coroutine-like procedures in programs that don't support such forms of control.

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

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 following sections demonstrate how to automatically transform a direct-style program into a CGI program, with no intervention from the programmer. While the transformation we describe relies on ideas taken from functional programming and compilers for functional languages, it can be used on programs written in languages that do not support any special functional-programming constructs, as we will demonstrate.

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 to develop Web programs.

Since CGI programs run in the context of a Web server, it is possible to write a custom server that allows CGI programs to behave as though they were interactive: it can provide CGI programs with re-implementations of primitives such as `display` or `prompt-read`, using a specialized version of `prompt-read` that uses Scheme's *call/cc* construct to capture the current control state as a continuation [40] value[3]. The server can associate this continuation with a new URL that accepts the inputs from a Web form and then store it for later resumption.

When the consumer submits a response to this Web form, the browser issues a request for the URL 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 the script as often as desired.

Prior work [26, 35] implements this approach and demonstrates its advantages. In addition to facilitating program construction, the modified Web

[3] A continuation value can be thought of as a function that when created captures the current list of computations left to perform before the program is complete, and when applied performs them.

server yields superior speed for CGI scripts compared to several existing methods.

Unfortunately, the approach has two severe problems. 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 theory, garbage collectors only reclaim memory that provably will never affect the rest of the computation. In practice, many languages provide weak references [37] (*i.e.,* references the collector ignores when determining reachability) so programmers can allow the collector to reclaim space sooner. The collector "proves" that values referred to only by weak references will never be used again by adjusting all the remaining weak references to some default value (e.g. false or *NULL*) indicating the value is gone.

Similarly, the Web server treats the URL references to continuations as somewhat weak. They are weak in the sense that the server will reclaim space sooner than the referring URLs disappear by redirecting those URL references to some default value—a Web page indicating the continuation is gone. The process differs from the usual notion of weak references by the criteria for breaking off the references. Instead of waiting until only weak references remain (which would eliminate the continuations immediately), the server relies on other criteria: timeouts.

Unfortunately, timeouts 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.[4]

- 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, because the garbage collector had already reclaimed the continuation, not even the site operator could grant an extension.

- On another occasion, one of the authors copied the first page generated by the registration program to a different file. Initial testing suggested

[4] 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.

that the copied page functioned correctly, yet the page stopped functioning a day later. The generated page had contained a link to the second page of a script (it should have re-started the script), so the bug only manifested itself when the timeout expired and the continuation was discarded.

3. Generating CGI Programs

The theoretical and practical problems with the server-based approach forced us to consider an alternative implementation technique. To simplify the presentation, we first demonstrate this technique on programs written in a purely functional subset of Scheme. In section 3.2, we extend it to support programs with state and mutation, and in section 3.3 we show how to apply the same technique to other languages without Scheme's advanced flow-control.

3.1. Functional CGI Programs

Removing timeouts would eliminate many of the problems encountered with our custom Web server. If the server could send the continuations to the clients, then the clients could decide how long to hold onto each continuation, eliminating the need for the server to cache the continuations and enforce a timeout policy. Accordingly, if we can marshal and unmarshal continuations into printable data, the server can send the marshalled continuations to clients (in a hidden field on the Web page) and the clients can send the marshalled continuations back to the server.

We employ three well-known transformations to enable the server to marshal the continuations. While these transformations were developed as techniques for compiling functional programs, they can be applied even to languages without the advanced features functional languages have, as we will see in section 3.3.

Continuation Passing Style (CPS) represents a program in such a way that at each point in its execution there is an explicit representation of everything still to be done before the program is complete [24]. In particular, each function of the program now consumes one additional argument: another function representing the continuation. In this style, a function that saves or manipulates continuations can simply refer to this new argument; in our case, a re-implementation of prompt-read can turn its new argument into a resumption point from which an external multiplexer can resume the program in response to a form submission.

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

(define c0
  (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))))))

(define c1
  (lambda (response1)    ;; the environment (in this case, holds
                         ;; the previous argument)
    (lambda (response2)  ;; the argument
      (display (+ response1 response2)))))

;; the converted functions and continuations
(define closures (vector c0 c1))

;; 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))

;; 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 '()))
```

Figure 4. The Compiled Version of Figure 1

Lambda lifting 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 [30].

Defunctionalization changes the representation of higher-order data, such as closures[5] and continuations, into a first-order form [38]. By choosing

[5] A closure can be thought of as an object with only one method: apply.

```
(define-struct closure (code env))

(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))))))

(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 5. The CGI Version of Figure 4 (Compare with Figure 2)

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.

These three phases are part of a standard technique for compiling functional languages ([24], [4]) first described by Reynolds [38]. CPS-converting, lambda lifting, and defunctionalizing partitions a program into separate interactive steps, so computation can halt conveniently between them and 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 4. This interactive program requires one final step to become a CGI program. The revision in figure 5 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.

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

```
(prompt-read-k "Enter the first number to add:"
  ;; lambda declares anonymous, first-class functions
  (lambda (res1)
    (prompt-read-k "Enter the second number to add:"
      (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-converted versions of higher-order primitives that accept (or return) call-backs must supply a continuation to their argument, since the callbacks may contain resumption points. External modules that accept function arguments must be transformed as well.

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

```
(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 4 shows the result of the final compilation step, namely of converting closures into structures; *apply-closure* performs function applications. 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 marshalled into strings for hidden fields and unmarshalled upon resumption. Since all closures have been converted into first-order closure structures, a function such as prompt-read can write a closure into the hidden field of a Web form and the CGI program can read this closure and apply it. Specifically, the code pointer of the continuation describes what subprogram to invoke next. The continuation's environment captures any values needed by the next subprogram instead of explicitly passing them in hidden fields.

Up to this point, the transformation produced a semantically equivalent program, so the result is a normal interactive program. 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 5. 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.

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

A drawback of this approach is that once a server starts any session using a particular secret encryption key, it cannot ever stop accepting sessions encrypted with that key without invalidating them. This flaw could make the security penalty for a compromised secret key worse.

3.2. COMPILING STATEFUL CGI PROGRAMS

```
(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 6. A Stateful Interactive Program

While generating CGI programs from interactive functional programs is almost a routine task with functional compilation techniques, *internal*[6] 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,[7] assignments to variables with assignments to

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

[7] Boxes are Scheme's mutable cells.

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

;; apply-closure : Closure (listof Value) * → Value
(define apply-closure ... ) ; as in figure 4
(define closures (vector ... ))

;; similar to figure 5
(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 7. CGI Version of Figure 6

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 6 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.[8] 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 7 sketches the cookie-based translation of figure 6.

Although this naïve cookie solution sounds straightforward, it has three 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 [32]. 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. A second minor problem is that modern browsers include many cookie-management facilities that expect Web sites to use cookies in very rudimentary ways, and some users disable cookies entirely as a security and privacy measure. This problem will presumably be minor, since users can always enable basic

[8] 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.

cookie management and refrain from manipulating cookies if they know they are interacting with a program that requires them. The third, 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 [36] 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.

3.3. Applying the Technique to Other Languages

The technique described in section 3.1 borrows heavily from techniques used to compile functional languages to machine code, but does not rely on the source or target language having any unusual features of functional programming languages: in particular, it does not require *call/cc* or other continuation-manipulation primitives and it does not require closures or first-class functions to be available; it requires only that the target language provide basic functions or a goto-like construct.

This may seem strange given that all phases of our technique refer to higher-order functions and at least CPS-conversion appears to rely on them. However, while the output of the compiler shown in figure 4 uses a few features of functional languages, it does not rely on them. To demonstrate this, we have provided an example of the same output as it might appear in C in figure 8.

Notice the similarity between figure 4 and figure 8. Other than a slightly more verbose syntax and a different format for the functions stored in the closure table (the Scheme version using curried environments, the C version taking environment and arguments at once) the programs strongly resemble one another.

```
#include <stdio.h>
typedef struct { int code ;  void *env ; } closure;
typedef void (*closuretype)(void *, void *);

closure *make_closure(int, void *);
void c0(void *, void *); void c1(void *, void *);
void apply_closure (closure *, void *);
void prompt_read_k(char *, closure *);

closure *make_closure(int code, void *env) {
    closure *k = (closure *) malloc(sizeof(closure));
    k→ code = code, k→ env = env;
    return k;
}
void c0(void *env, void *response1) {
    closure *k = make_closure(1, response1);
    prompt_read_k("Enter the second number to add: ", k);
}
void c1(void *response1, void *response2) {
    printf("%d\n", (int) response1 + (int) response2);
}

closuretype closures[] = {c0,c1};
void apply_closure (closure *f, void *args) {
    (*(closures[f→ code]))(f→ env, args);
}

void prompt_read_k(char *s, closure *k) {
    char input[10];
    int i;
    printf("%s", s);
    fgets(input,10,stdin);
    i = atoi(input);
    apply_closure(k,(void *) i);
}

int main() {
    closure *k = make_closure(0, (void *) 0);
    prompt_read_k("Enter the first number to add: ", k);
    return 0;
}
```

Figure 8. The C version of Figure 4

```
void apply_closure(closure *clo, void *args) {
    int code = clo→ code;
    void *env = clo→ env;
    free(clo);
    (*(closures[code]))(env,args);
}
```

Figure 9. Modification of Figure 8 to reclaim continuation frame memory

Memory Management

Figure 8 mirrors figure 4 closely, but has one important difference: because C requires programmers to manage memory manually while Scheme reclaims memory automatically, the memory allocated for closures using *make_closure* in the C program is never reclaimed. This is easy to fix: since closures are only ever introduced to this program by the CPS-conversion phase and are never live after their first use, we can simply free them when we apply them by adding a call to free to *apply_closure*, as shown in figure 9.

Freeing closures at this point will work with any C program compiled with our compiler since source C programs will never create first-class closures of their own.

Eliminating malloc

The previous section demonstrated that the memory our compiler allocates for continuation frames can be freed reliably, but programmers concerned with efficiency may object to using the heavyweight malloc and free method of allocating closures for something as fundamental as managing stack frames. malloc is a very slow procedure compared to a stack push and free is slow compared to a stack pop, so large programs might suffer greatly from using them. Happily, since allocation and deallocation of continuations in this program (and any program generated from C source using this technique) will occur in a strict stack, we can rewrite *make_closure* to avoid using malloc and free entirely, instead using *push* and *pop* (figure 10).

Eliminating Stack-Based Memory Leaks

Unlike Scheme, C is not properly tail-recursive [11], meaning that when a function calls another function as the last operation it performs, such as *c0* does in figure 8. C does not reclaim the memory allocated to its stack frame until the function it called in tail position returns, so in the instance of *c0* its stack frame will not be collected until *prompt_read_k* returns, even though in

```
int curr;
closure stack[MAX_STACK_SIZE];

closure *push(int code, void *env) {
    closure *ret = &stack[curr];

    ret.code = code;
    ret.env  = env;

    ++curr;
    return ret;
}

void pop() {
    --curr;
}
```

Figure 10. Modification of Figure 9 for faster stack behavior

principle that memory contributes nothing to the computation after *c0* calls *prompt_read_k*.

In many C programs following normal C programming style, this limitation does not pose a serious problem; C programmers simply avoid making long chains of tail-calls. However, the efficiency of continuation-passing style relies on efficient tail calls, so we have yet another memory-related problem to solve.

Fortunately, there are many ready solutions. The easiest in C would be to add two additional global variables, *closure* $*$*clo_reg* and **void** $*$*args_reg*, rewrite all continuation functions to be labelled code segments, and use **goto** instead of tail calls entirely, maintaining a stack only when necessary to determine where a future **goto** will lead. But this technique, which in fact takes the program most of the way to assembly code, seems too specific to C to convince the skeptic reader that our technique applies to languages without special features; and besides, other simple techniques are available.

One such simple technique is called *trampolining*.[9] In this method, rather than calling *apply_closure* directly, each closure allows an outer "trampoline" loop sitting at the bottom of the stack to do so. To implement trampolining, we would again add *closure* $*$*clo_reg* **void** $*$*args_reg*. Then, we would rewrite

[9] We choose trampolines here because they are easy to implement and have reasonable performance characteristics. Those interested in implementing a direct-style CGI compiler for C should see [6] for an alternative technique that reuses stack frames as live memory.

```
void prompt_read_k(char *str, closure *k) {
    char input[10];

    /* ...other code as before */

    i = atoi(input);
    clo_reg = k;
    args_reg = (void *) input;
    return;
}

int main() {
    /* ...as before ... */

    while (1) { apply_closure(clo_reg, args_reg); }
}
```

Figure 11. Using trampolining to avoid stack explosion

each call to *apply_closure* as we have with *prompt_read_k* and augment our main function to finish with a trampoline loop, as shown in figure 11.

With that, we can finally eliminate all our memory difficulties and have an automatically-restructured C program compile to an efficient executable that only needs boilerplate as in figure 5 to be a full-fledged CGI program.

Eliminating Function Pointers

We used the C example above to make the argument that our technique would work with any language that supported functions, but in that example we made use of C function pointers, a language feature specific to C. While languages that have no provision for any kind of calculated jump (*e.g.*, function pointers, closures, or objects) are rare, our technique does not require calculated jumps at all. To eliminate them from our C example, we could simply merge *apply_closure* and the *closures* table into a single function:

```
void apply_closure (closure *clo, void *args) {
    switch (clo-> code) {
        case 0: c0(clo-> env, args); break;
        case 1: c1(clo-> env, args); break;
    }
}
```

That modification would eliminate all uses of function pointers, making the code entirely free of calculated jumps; translation to other languages or assembly code from this point is straightforward.

TRANSLATION PHASES

The output, then, can be expressed in languages without functional programming features. The translation phases can also be applied to languages that do not support those features: though our intermediate phases make heavy use of higher-order functions, the intermediate representation of the program being transformed does not need to be directly executable. So, were we transforming C or another language without closures, the CPS-conversion phase could produce its output in C augmented with closures, knowing that every closure we introduced in this phase would be eliminated in the defunctionalization phase thus making the compiler's final output a legal C program. The idea that a CPS transformation followed by closure conversion, lambda lifting and defunctionalization can be combined into a simpler transformation was first noticed in [18].

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 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

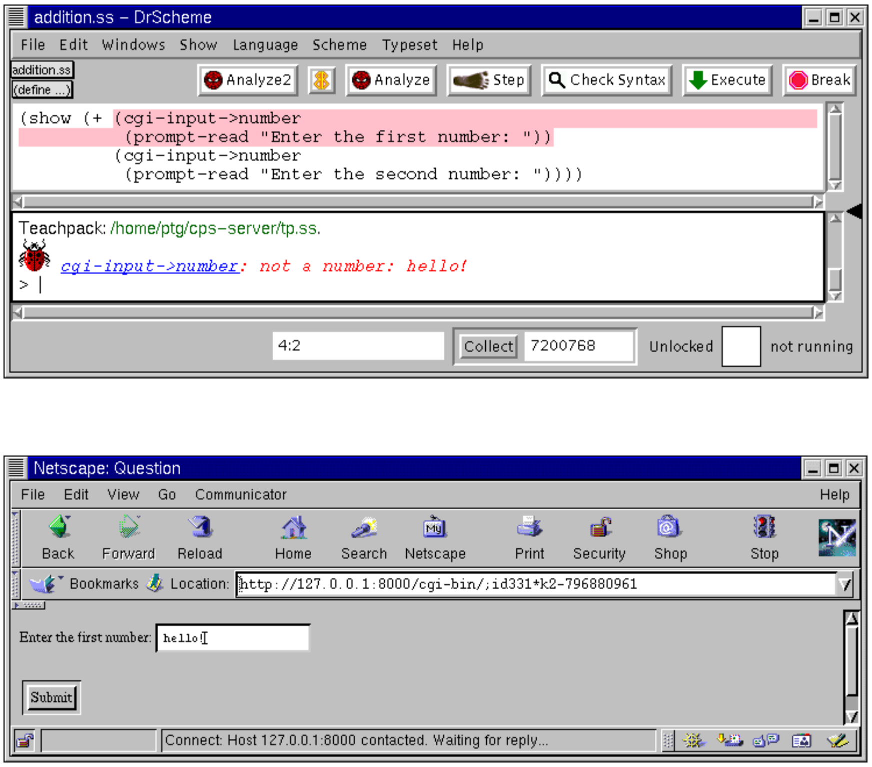

Figure 12. CGI Error Reporting

communicates the submission of a Web form to these primitives. Furthermore, the new library keeps track of the continuations of prompt-read so that the developer can truly simulate a consumer's actions on the browser.

To demonstrate this idea, we wrote a library (technically, a Teachpack [16]) of interaction functions for DrScheme, our programming environment [16] 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)

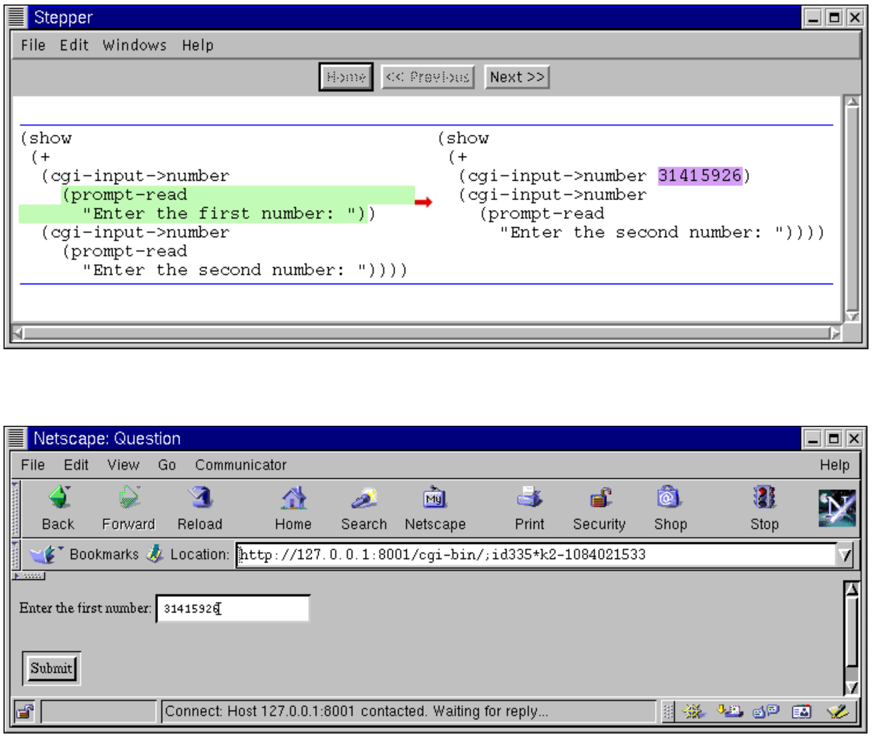

Figure 13. CGI Stepping

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 12 for an illustration.

Consider the more complex example of DrScheme's single-step debugger [10]. 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 13 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

We first developed a prototype CGI compiler that operates on core Scheme programs. The compiler accepted a single expression, typically a **letrec** expression prefixed with (uncompiled) PLT Scheme library specifications. We used the prototype to test the feasibility of the compiler with a number of examples, plus one full-fledged application: the teacher enrollment dialog for our TeachScheme! outreach project.

Our effort to turn the prototype compiler into a full-fledged compiler for all of MzScheme [19] is ongoing.

MzScheme extends the Scheme R^5RS [31] standard with several features, including continuation marks [10], units [21], mixins [17, 23], and modules [20]. Many of these extensions, including units and mixins, are already compiled to a core λ-based language by MzScheme's front end. Others, notably modules, are not, since there is no equivalent for them in the core λ-based language. Accordingly, our compiler must be able to CPS convert these forms, without re-implementing them. In general, this is not possible, but we have been able to extend MzScheme in simple ways to cope with all of its extensions.

Modules

Once MzScheme's front end has processed a module, that module consists of a series of function definitions and expressions plus declarations about which other modules it imports and which identifiers it exports to other modules. To CPS-convert it, we transform the module so that it exports one additional function: *init*. This new function accepts no arguments and performs all of the initialization code from the original module, including calling the *init* functions from imported modules. Since the transformed module only has function definitions, we can CPS convert the entire module by CPS converting each definition.[10] To make this transformation work properly, we had to add a new renaming mechanism to the module system.

Operating System Resources

MzScheme lets programs access several operating system resources such as TCP/IP connections and files. When a prompt-read occurs, since the Web

[10] This technique is sufficient for most languages, but since our compiler is written for Scheme, we go further. In Scheme, functions definitions are just like variable definitions initially bound to a λ expression, and recursive binding constructs are semantically identical to non-recursive binders, initialized to a dummy value, followed by a series of assignments [31]. Accordingly, our compiler pushes the initializations of all of the module's definitions into the *init* function and then CPS converts it.

server will terminate MzScheme and thus force it to relinquish these resources, we must decide what will happen to them. While we would like to preserve them so that programmers could treat Web interaction as any other kind of user interaction, this is not possible in the general case. For example, the resources could be a network connection to an interactive process on another machine that could time out. One can imagine migrating these resources to the client machine and back, but rather than try that, we elect to implement the simpler strategy of just releasing all such resources.

Since programs running under the compiler terminate the resources when prompt-read is called, the program development environment must be able to mimic this behavior. We use MzScheme's custodians [22] to close all open TCP/IP ports and files (as well as all of the other OS resources that the program may have allocated).

THREADS

While as discussed in the previous section we cannot preserve operating system resources across Web interaction lines in general, we can preserve one particular operating system resource: threads of control.[11] MzScheme supports preemptive threads, created by the *thread* primitive. It accepts a function of no arguments, creates a new thread and applies the function on the new thread. As with other features related to operating system resources, this feature poses a problem for the CGI compiler: a program could start several threads and then request user input, which due to CGI restrictions shuts down the entire application including all threads.

To cope with this, we could take the default strategy of the previous section and shut down all threads. For new applications written specifically for the Web, that constraint probably would not be a too burdensome most of the time. However, stopping threads seems like a big problem for maintainers of legacy software, so we have decided to allow threads to appear to remain alive across Web interactions by stopping and marshalling all threads when a prompt-read occurs, and resuming all running threads when the computation resumes.

To implement this feature, we use a single assignable cell per thread. The CGI compiler redefines the *thread* primitive to allocate a new cell. At each application of a continuation, the transformed program updates the cell for the current thread with the new continuation. When a prompt-read occurs, we interrupt each running thread, gather the continuations from all of the cells, and marshal all of them into the Web page that we send to the browser.

[11] Technically, the MzScheme interpreter manages threads itself without relying on the operating system's underlying thread-related features. However, the technique for managing threads would remain unchanged in either case.

It is worth noting that the resulting datum could potentially be quite large if the program has many threads active when it calls prompt-read. Also, a legacy program might be using threads while requesting user input specifically for the speedup that comes with parallelism, a speedup we will have to deny it. However, our system imposes no constraints that aren't inherent to the problem, and preserves the semantics of threads so developers can quickly migrate a program to the Web and then incrementally improve its performance.

CONTINUATION FRAMES

The transformation outlined in the paper creates a separate continuation application for each subexpression in the program, even those that will never be split across a prompt-read. This significantly increases the size of the compiled program and also increases the size of the continuations sent to the browser.

In principle, our compiler does not need to create a separate continuation for each subexpression, since prompt-read only happens at well-defined points in the evaluation. Instead, we could identify the functions that are guaranteed not to call prompt-read during their dynamic extent. The bodies of such functions do not have to be converted to CPS form, since either the entire function will appear in the continuation or the function will not appear in the continuation at all.

Unfortunately, this optimization interferes with our strategy for supporting multithreaded programs described in the previous subsection because that strategy requires that we be able to interrupt any thread at an arbitrary point and retrieve its current continuation, which will not generally be available if we employ this optimization. Negotiating this conflict is an important direction in our ongoing research.

CALLBACKS FROM PRIMITIVES

Many of MzScheme's primitive functions are implemented in C, as part of MzScheme's runtime system. Several of these C language procedures invoke arguments that are Scheme procedures, triggering a callback from the C code to Scheme code. The Scheme standard includes a few of these (map, for-each, etc) and MzScheme includes several more (andmap, ormap, user-defined ports, etc).

In order to CPS convert programs that use those primitives, we must CPS convert the primitives themselves. Since our compiler only consumes and produces Scheme, we have to either re-implement these primitives or build a CPS converter for C code. We have chosen to re-implement the primitives in Scheme. As an example, Scheme's map is replaced with this function:

(**define** (*map/k f l k*)

```
(let loop ([l l] [k k])
  (cond
    [(null? l) (k '())]
    [else (f (car l)
            (lambda (hd)
              (loop
                (cdr l)
                (lambda (tl)
                  (k (cons hd tl))))))]))
```

MARSHALLING

MzScheme introduces many new kinds of values: hash-tables, structures, custodians [22], and parameters, to name a few. MzScheme does not provide marshalling and unmarshalling (*i.e.*, saving values to disk) for most of these values so, as part of our compiler, we need to provide marshalling and unmarshalling. For many of the values, *e.g.*, hash-tables, MzScheme already supports enough operations to implement marshalling and unmarshalling. For others, *e.g.*, custodians, our compiler must replace all of the operations on the value with new versions of the primitives that record enough information to be able to save the values to disk and restore them.

6. Related Work

Programmers building imperative-style programs in purely functional languages use a technique based on the mathematical theory of monads. Hughes [27], 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 [35] 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, limit the utility of 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 [25] claims that the success of his Viaweb company, now Yahoo! Store, 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 [34] 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 `<bigwig>` system [5, 9] 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 [12] 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. The Web server, however, 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 [41] enables Java development environments [8, 28, 42] 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.

Thiemann [43] used Hughes's ideas as a starting point and provides a monad-based library for constructing Web dialogs. His monads take care of the "compilation" of Web scripts into a suitable continuation form. Working with Haskell, Thiemann can now use Haskell's type system to check the natural communication invariants between the various portions of a Web program. Haskell, however, is also a problem because Thiemann must accommodate effects (interactions with file systems, data bases, etc) in an unnatural manner. Specifically, for each interaction, his CGI scripts are re-executed from the beginning to the current point of interaction. Even though this avoids the re-execution of effects, it is indicative of the problems with Thiemann's approach.

7. Conclusion

Our paper introduces an automated translation that implements an interactive programming model for Web applications. By helping the programmer avoid having to manually save and restore control state between interactions, the system not only eases the initial software development, but also facilitates maintenance, since the CGI programming model now matches the traditional interactive programming model. The matched models also ensures that software engineers can still use familiar programming tools when doing Web development. As an example, the paper demonstrates how our technique allows developers to continue using 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. Furthermore, we can implement these transformations for almost any modern high- or low-level language by either using the language's built-in language features for an executable intermediate representation or by introducing and immediately eliminating closures. Most encouragingly, our work shows that it is possible to implement our transformation in such a way that we can preserve most of the semantics of a full-featured programming language while adding unrestricted ability to interact with Web users.

One important question we leave unresolved is how to decide which values should go in the store and which should go in the environment. For instance, in C, loops usually exit when an index variable is destructively updated to a particular value, but C programmers might be surprised to find that returning to the middle of a loop does not reset the loop counter to the value it previously held. Fundamentally, the Web permits new observations on the behavior of the program that were not possible before. Concretely, a loop implemented with *for* in C that imperatively updates an index variable and a loop implemented with a recursive function in C[12] are indistinguishable in the traditional interactive model, but are distinguishable with Web programs. Still, our research strongly suggests that our technique can be used as it is to automatically restructure programs written in a wide variety of languages.

ACKNOWLEDGEMENTS

We thank Morgan McGuire, Greg Pettyjohn, the Automated Software Engineering 2001 anonymous referees, and the Journal of Automated Software Engineering 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. Appel, A.: 1992, *Compiling With Continuations*. The Pitt Building, Trumpington Street, Cambridge CB2 1RP: Cambridge University Press.
5. Atkins, D. L., T. Ball, G. Bruns, and K. C. Cox: 1999, 'Mawl: A Domain-Specific Language for Form-Based Services'. *Software Engineering* **25**(3), 334–346.
6. Baker, H. G.: 1994, 'CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A.'. comp.lang.scheme.c newsgroup.
7. Bakken, S. S., A. Aulbach, E. Schmid, J. Winstead, L. T. Wilson, R. Lerdorf, A. Zmievski, and J. Ahto: 2002. `http://www.php.net/manual/`.
8. Borland Software Corporation. `http://www.borland.com/jbuilder/`.
9. Brabrand, C., A. Møller, A. Sandholm, and M. I. Schwartzbach: 1999, 'A Runtime System for Interactive Web Services'. In: *Journal of Computer Networks*.
10. Clements, J., M. Flatt, and M. Felleisen: 2001, 'Modeling an Algebraic Stepper'. In: *European Symposium on Programming*.
11. Clinger, W. D.: 1998, 'Proper tail recursion and space efficiency.'. In: *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 174–185.
12. Coward, D.: 2000, 'Java Servlet Specification Version 2.3'. `http://java.sun.com/products/servlet/`.
13. Daemen, J. and V. Rijmen, 'Advanced Encryption Standard'. `http://www.nist.gov/aes`.
14. Felleisen, M. and R. Hieb: 1992, 'The revised report on the syntactic theories of sequential control and state'. *Theoretical Computer Science* **102**, 235–271. Original version in: Technical Report 89-100, Rice University, June 1989.

[12] assuming the compiler supports the tail-call optimiziation

15. Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee: 1999, 'Hypertext Transfer Protocol – HTTP/1.1'. Internet Request for Comments 2616.
16. Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen: 2002, 'DrScheme: A Programming Environment for Scheme'. *Journal of Functional Programming* **12**(2), 159–182. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.
17. Findler, R. B. and M. Flatt: 1998, 'Modular Object-Oriented Programming with Units and Mixins'. In: *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. pp. 94–104.
18. Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen: 1993, 'The Essence of Compiling with Continuations'. In: *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, Vol. 28(6). New York: ACM Press, pp. 237–247.
19. Flatt, M.: 1997, 'PLT MzScheme: Language Manual'. Technical Report TR97-280, Rice University. `http://www.plt-scheme.org/software/mzscheme/`.
20. Flatt, M.: 2002, 'Composable and Compilable Macros: You Want it *When?*'. In: *Proceedings of ACM SIGPLAN International Conference on Functional Programming*.
21. Flatt, M. and M. Felleisen: 1998, 'Units: Cool Modules for HOT Languages'. In: *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 236–248.
22. Flatt, M., R. B. Findler, S. Krishnamurthi, and M. Felleisen: 1999, 'Programming Languages as Operating Systems (*or* Revenge of the Son of the Lisp Machine)'. In: *Proceedings of ACM SIGPLAN International Conference on Functional Programming*. pp. 138–147.
23. Flatt, M., S. Krishnamurthi, and M. Felleisen: 1998, 'Classes and Mixins'. In: *Proceedings of the ACM Conference Principles of Programming Languages*.
24. Friedman, D. P., M. Wand, and C. T. Haynes: 1992, *Essentials of Programming Languages*. Cambridge, MA: The MIT Press.
25. Graham, P., 'Lisp in Web-Based Applications'. `http://www.paulgraham.com/lib/paulgraham/bbnexcerpts.txt`.
26. Graunke, P., S. Krishnamurthi, S. van der Hoeven, and M. Felleisen: 2001, 'Programming the Web with High-Level Programming Languages'. In: *European Symposium on Programming*.
27. Hughes, J.: 2000, 'Generalising monads to arrows'. *Science of Computer Programming* **37**(1–3), 67–111.
28. International Business Machines, Inc. `http://www.ibm.com/websphere`.
29. Jackson, M. A.: 1975, *Principles of Program Design*. Academic Press.
30. Johnsson, T.: 1985, 'Lambda lifting: transforming programs to recursive equations'. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. Nancy, France.
31. Kelsey, R., W. Clinger, and J. R. (Editors): 1998, 'Revised^5 Report of the Algorithmic Language Scheme'. *ACM SIGPLAN Notices* **33**(9), 26–76.
32. Kristol, D. and L. Montulli: 1997, 'RFC 2109: Proposed Standard for HTTP State Management Mechanism'. http://www.ietf.org/rfc/rfc2109.txt.
33. NCSA, 'The Common Gateway Interface'. `http://hoohoo.ncsa.uiuc.edu/cgi/`.
34. Open Market, Inc., 'FastCGI specification'. `http://www.fastcgi.com/`.
35. Queinnec, C.: 2000, 'The influence of browsers on evaluators or, continuations to program Web servers'. In: *ACM SIGPLAN International Conference on Functional Programming*.

36. Reed, D. P.: 1983, 'Implementing atomic actions on decentralized data.'. In: *ACM Transactions on Computer Systems*. pp. 234–254.
37. Rees, J. A., N. I. Adams, and J. R. Meehan: 1988, 'The T Manual'.
38. Reynolds, J.: 1972, 'Definitional Interpreters for Higher-order Programming Languages'. In: *Proceedings of the 25th ACM National Conference*. pp. 717–740.
39. Roth, M. and E. Pelegrí-Llopart: 2002, 'JavaServer Pages Specification'. `http://java.sun.com/products/jsp/`.
40. Strachey, C. and C. P. Wadsworth: 1974, 'Continuations: A Mathematical Semantics for Handling Full Jumps, Technical Monograph PRG-11'. Technical report, Oxford University Computing Laboratory, Programming Research Group.
41. Sun Microsystems, Inc. `http://java.sun.com/products/jpda/`.
42. Sun Microsystems, Inc, 'Forte Tools'. `http://www.sun.com/forte/`.
43. Thiemann, P.: 2002, 'WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms'. In: *Practical Applications of Declarative Languages*.