Higher-Order and Symbolic Computation, 17, 277–295, 2004

Continuations and Web Servers

CHRISTIAN QUEINNEC Christian.Queinnec@lip6.fr
Université Paris 6, Pierre et Marie Curie, LIP6, 4 place Jussieu, 75252 Paris, Cedex 05, France

Abstract. Programming web applications in direct style with the help of continuations is a much simpler, safer, modular and better-performing technology than the current dominating "page-centric" technology combining CGI scripts, active pages or servlets. This paper discusses the use of continuations in the context of web applications, the problems they solve as well as some new problems they introduce.

Keywords: scheme, continuations, hypertext, WWW

1. Introduction

Web technologies are ubiquitous. Browsers are the interface of choice for all kinds of interactions with remote services. Browsers provide a small but quite portable graphical toolkit (windows, menus, buttons, text rendering, etc.) as well as scripting facility (JavaScript). Server applications may then rely on "thin clients" to which they delegate tight interactions with users (scrolling, input editing, clicking, etc.) There are at least two major differences between a browser and a regular graphical (GUI) toolkit:

1. The relationship between servers and clients is asymmetric: servers cannot spontaneously update clients, they only respond to clients' requests;
2. browsers provide additional facilities such as bookmarking, going back in their history of interactions, cloning windows, etc. with the net effect that servers have no control whatsoever on exact clients' states.

A web application is an interactive application that runs on a web server whose access medium is a browser sending requests and receiving responses over HTTP. A web application is often viewed as a series of (programs leading to) pages (or screens in more traditional parlance) delivering some information and asking for some new information to further the dialog. When a server sends such a page, it stops and waits for further information. When this information becomes available to the server, the server resumes its work from that point. Thus server and client act as co-routines. The key point of this paper is to explain this model in terms of continuations.

In a web application, the process of the server waiting for further information from the client and then resuming its work may be simply explained as the server letting the client know its current continuation within the application, with the additional provision that subsequent clients' requests tell which continuation the server should resume.

Note that, depending on the point of view, I will use the client-server terminology where clients emit *requests* towards the server in order to get *responses*. From an opposite

point of view, the web application *asks* the user for some information and the user's *answer* resumes the web application. Due to the underlying technology, answers are communicated via requests while questions are communicated via responses. The first request initiates the web application.

From this point of view centered on continuations, jumping back into the browser history is a means of reinvoking an already invoked continuation, that is, for the client, to answer to an already answered question. Cloning a window is a means of invoking a continuation more than once, that is, to answer (quasi simultaneously and possibly differently from both windows) a question more than once.

After some historical reminiscences explaining the genesis of these ideas in Section 2, Section 3 will present an essential implementation of the server-side framework. In Section 4, a very simple web application shows the benefits of this approach from the client-side. Concurrency being a by-product of browser technology is addressed in Section 5, serializing continuations is discussed in Section 6 while a scoping problem is solved in Section 7. Related works and conclusions end the paper.

2. History

In December 1998, I was devising an educational CD-ROM [27] whose topic was the C programming language that I was teaching at UPMC. The CD-ROM contained various lecture notes and exercises as well as a number of documents related to C (Guides of style, FAQ, historical documents, copies of web sites devoted to C, etc.) The CD-ROM was equipped with its own web server (based on Tomcat [3]) allowing students to browse the dynamic pages of the CD-ROM.

I wanted to propose "trails" to students (read some pages, work on some exercises) while providing them some links to suggested readings. I wanted these trails to be downloadable from the University web site so I could adapt them long after the CD-ROM was burnt. The problem was: how could I give to the students an immediate way to come back to the trail after (encouraged) wandering?

The idea was to consider the trail as a program. When wandering, the CD-ROM web server will embed within all served pages a button bound to the continuation of the trail where the wandering started. Clicking the button would resume the associated continuation and put the student back on the trail. All that was left to do was to choose a language with reifiable continuations for the trail, to bind these continuations to URLs when serving pages and to instruct the server to invoke continuations when serving these URLs.

To elect Scheme was then a simple matter as it was the sole standardized language offering continuations as first-class values, to choose an interpreter allowed trails (Scheme S-expressions) to be easily downloadable and to write this interpreter (named PS3I) in Java was a simple task [26]. However, to graft this interpreter into a web server was not so simple due to the unavoidable coexistence of continuations and concurrency (cf. Section 5.3).

3. Serving with continuations

As excellently mentioned by Graunke et al. [15], a web server is akin to an operating system that receives HTTP requests and runs programs to generate the appropriate (static or dynamic) responses.

If the web application needs some information, it has to ship a page towards the client. This page is usually an HTML form with a submit button that will trigger the resumption of the web application. The framework provides the `show` function to ship pages. This function takes a single argument (named `page-maker` in the following definition), a function that produces a string of HTML or whatever variant (XHTML). The `page-maker` function consumes an argument, a string representing the URL that will become the value of the `action` attribute of the `<form action=...>` HTML tag. When requested, this URL will trigger the resumption of the web application. Here is a definition of `show`:

```
; (URLstring → HTMLstring) → HTTPRequest
(define (show page-maker)
  (call/cc
   (lambda (resume)
     (let ((resumption-url (register-continuation! resume))
           (connection (current-connection)) )
       (display (page-maker resumption-url) connection)
       (close-output-port connection)
       (suicide) ) ) ) )
```

Before anything else, the `show` function grabs its continuation using the predefined Scheme `call/cc` operator, registers that continuation and, in return, obtains a URL (a string here named `resumption-url`) associated with that continuation. This resumption URL is then given to the `page-maker` function to produce the complete HTML page which is displayed onto the current TCP connection. The TCP connection is then closed. After the page is shipped, the thread servicing the current request commits suicide and disappears thus freeing some resources (connection, thread, memory, etc.) managed by the web server.

For now we can assume that registering continuations is simply performed with some global hashtable. When the connection is opened in HTTP/1.0 mode, the `close-output-port` function effectively closes the TCP connection. When in HTTP/1.1 mode, `close-output-port` only resets the connection so it can convey further requests.

The resumption URL may be represented by a string or by a more complex data structure. There is more than one way to name a resumption URL to retrieve a continuation, here is probably one of the simplest (though obfuscating, signing or ciphering, this URL would be recommended if one wants to avoid clients forging URL in order to steal others' continuations):

```
http://my.server/resume?continuation=813
```

When the web server is requested, it parses the request and reifies it into a data structure (for instance, an `HTTPRequest` as for Java servlets [9]) then it creates a thread to service that request. The `service` function may be defined as:

```
 ;  HTTPRequest → Nothing
(define (service request)
  (let ((k (get-registered-continuation
             (get-request-url request) )))
    (if k
        (k request)
        (error "Missing continuation") ) ) )
```

The `service` function extracts the URL from the HTTP request with the function `get-request-url`, uses it as a key to retrieve the bound resumption continuation; in other words, `get-registered-continuation` is a left inverse of `register-continuation!`. If a continuation is obtained, it is invoked with the HTTP request data structure as argument. Due to the mechanism of continuations, this HTTP request will become the value of the call to the `show` function that registered that precise continuation.

From the web computation point of view, it looks as if the `show` function ships out a page, waits until a request for its continuation arrives and returns as result this request value. Possible implementations may block a thread (or process) as in fastCGI [22] or <bigwig> [4] or, two threads (or processes) may be used: one thread ships the page, another thread resumes the web computation, this new thread starts in the precise control state where the previous one stopped. These two threads are linked by the continuation of the call to `show`.

4. A tiny web application

Let us give a very simple but complete example of a web computation. The "Sum Web Application" just sums numbers. The top row of Figure 1 illustrates its regular behaviour.

The Sum Web Application first requires a number then proposes to complete a partial addition whose first operand is the first number entered. When a second number is entered, the sum is displayed and an additional number may be further submitted for larger and larger additions. In Figure 1, the numbers that are input appear on top of the arrows linking successive screens.

My claim is that the results shown in the web pages must be independent of the order in which the pages were traversed. For instance, the user may start with the home page (the screen at the top left of Figure 1) input 11, 22, then 44, then hit the "Back" button twice, input 33, hit "Back" twice again and finally input 101 followed by 202. The same screens are obtained if the user clones the home page, inputs 11 in the first window, switches to the second window and inputs 101 then 202, switches back to the first window, inputs 22, bookmarks the resulting page, hits "Back", inputs 33, goes to previously bookmarked page and inputs 44.

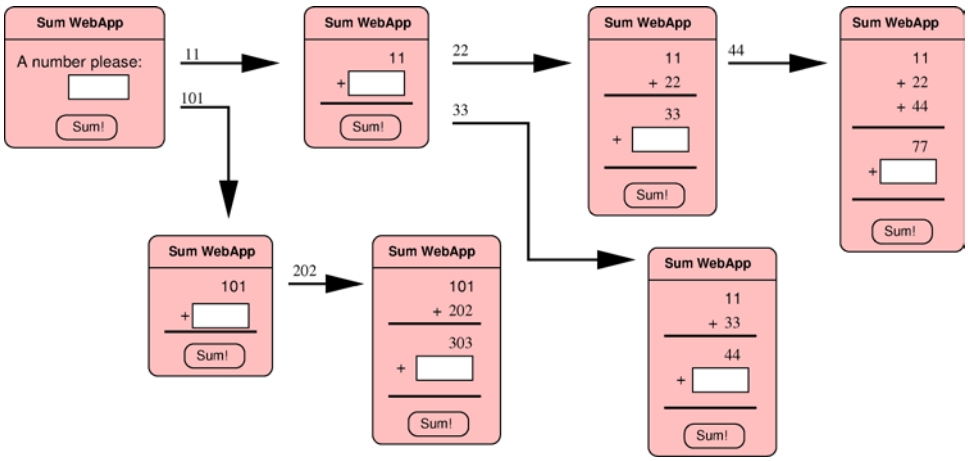

Figure 1. The sum web application.

```
; String
(define *RESPONSE-FIELD* "number")

; (URLstring → HTMLstring) → number
(define (read-number page-maker)
  (let ((request (show page-maker)))
    (or (string->number
         (get-request-post-parameter request *RESPONSE-FIELD*) )
        (read-number page-maker) ) ) )
```

Figure 2. Helper function.

One helper function for the Sum Web Application appears on Figure 2, the two main functions (one is specialized for the HTML generation, the other holds the logic of the web application) appear on Figure 3. The whole application is roughly fifty lines long.

The expression `(sum-web-application)` starts the web application. The logic of the web application fits entirely within that function. It recursively asks for a number and appends it to the list of numbers so far acquired. The page sent to ask for a number is generated by the `generate-addition` function that is dedicated to HTML generation. According to the length of the argument (the list of numbers accumulated so far) it displays an input text field to require a first number or it displays the addition of the accumulated numbers followed by an input text field proposing to submit yet another number. The HTML generation is performed *à la LAML* [21] with functions bearing the names of the equivalent HTML tags. The whole structure looks similar to HTML and should pose no problem of comprehension, see Figure 1 for a sketch of these HTML renderings.

The user's answer is handled with the help of the `read-number` function which loops until it receives a valid number, that is, a string that may be correctly converted into a number. The `read-number` and `generate-addition` functions must use the same name for the `name` attribute of the `<input type='text' ...>` tag. This common name is

```
; List[number] → (URLstring → HTMLstring)
(define (generate-addition numbers)

  (define (generate-rows numbers)
    (define (generate-other-rows numbers)
      (if (pair? numbers)
          (div (tr (td align: "right" "+" (car numbers)))
               (generate-other-rows (cdr numbers)))
          (tr (td (hr))) ) )
    (div (tr (td align: "right" (car numbers)))
         (generate-other-rows (cdr numbers)) ) )

  (lambda (url)
    (html (head (title "Sum WebApp"))
          (body
           (form method: "POST" action: url
                 (if (null? numbers)
                     (p "A number please: "
                        (input type: "text"
                               name: *RESPONSE-FIELD* ) )
                     (table
                      (if (>= (length numbers) 2)
                          (generate-rows numbers)
                          "" )
                      (tr (td align: "right" (apply + numbers)))
                      (tr (td align: "right"
                              "+" (input type: "text"
                                         name: *RESPONSE-FIELD* ) ))
                      (tr (td (hr))) ) )
                 (input type: "submit" value: "Sum!") ) ) ) ) )

; Nothing → number
(define (sum-web-application)

  (define (loop numbers-so-far)
    (let ((next (read-number (generate-addition numbers-so-far))))
      (if (= next 0)
          (apply + numbers-so-far)
          (loop (append numbers-so-far (list next))) ) ) )

  (loop '()) )

; Start the web application:
(sum-web-application)
```

Figure 3. Sum web application.

the value of the global `*RESPONSE-FIELD*` variable. Type-based analyses to ensure this consistency may be found in Thiemann's WASH/CGI system [32], Graunke et al. [19] or JWIG [6].

Observe that continuations do not appear explicitly in this web application: the web application programmer does not have to worry about continuations, invoking the `show` function and understanding its behaviour is sufficient. The Sum Web Application is said

to be written in "direct style". More elaborate libraries such as a library for quizzes [29] may be devised since quizzes require a tighter management of continuations for example to refrain students from giving all possible answers for a question offering a finite choice or to change their answers to former questions when discovering new questions.

Observe also that the whole web application (and its different types of pages) is defined in a single file making it easy to understand the logic of the dialog as a whole before, say, the details of the HTML generation or the arcane transmission of information between pages.

The resumption is entirely automatic since continuations and closures are the two linguistic features that take care of preserving the control state and the lexical environment to be reinstated. If your favorite language does not have these features, you can "compile" your web application by hand along the lines set by Graunke et al. [14] that is, apply CPS (Continuation-Passing Style) to remove continuations then apply defunctionalization to remove closures and higher order functions. All programming languages may then run the resulting program.

When a web server receives a request, it analyzes it and tries to figure which user sent it. If this user can be determined, the web server associates it to a hashtable. This hashtable is made available to the targeted page as the "Session object" (in ASP, PHP or servlet parlances [9]). If the targeted page records values in this Session object, these values will be available to the next page the user will request. The implementer of a web application then needs to pass information from page to page via the Session object. Without closures and continuations, the implementer needs to discover what exactly is to be resumed (on the control side) and what data is required to resume that work. Since the Session object is shared by all resumption points, it strongly discourages the programmer from taking care of any requests but the one that is (normally) expected. And quite often as noted by Graunke et al. [19] this is not even checked!

Web applications in direct style are modular; they may be developed independently and then freely combined. The continuation is not restricted to the sole context of the web application, it captures the entire control state. In the Sum Web Application, the `read-number` function might be viewed as a small parameterizable web application whose goal is to ask for a number and return that number. For that goal, `read-number` contains some logic to loop in case of bad inputs.

Given that the Sum Web Application stops and returns the accumulated sum when zero is submitted, the whole web application can be viewed as a (admittedly contorted) means to choose a number. That result may then be fed into another web application as in:

```
(other-web-application (sum-web-application))
```

Continuations captured while processing the inner Sum Web Application grab the full control state including the embedding call to the "Other Web Application". The code of the Sum Web Application is thus totally unaffected by its context of use. To use Session objects would be harder since Session objects must be managed cooperatively by the web applications because they are shared.

It is therefore possible that the inner web application returns more than once with various values thus allowing the user to adopt a "what if" strategy that is, inputting a number and

looking at the obtained results then hitting the "Back" button and inputting an alternate number to check for the differences and so forth.

5. Continuations and concurrency

The standardized definition of Scheme [17] offers continuations as do many implementations of other languages (SML-NJ for instance). Recently, designers of the Cocoon framework for Java enhanced their JavaScript engine (named Rhino) to provide first-class continuations [25]. However, the mere availability of continuations is not sufficient. Continuations do come with their own set of problems. This section discusses some of them and then proposes to refine the notion of interaction within a top-level loop.

Browsers induce concurrency within servers! For instance, one may clone a window whose content is a question and then may answer this same question differently but simultaneously from the two windows. This is even scriptable with a few lines of JavaScript (see [28] for detail). Two threads within the server are now processing (possibly in parallel) the two different answers for the same question. In contrast to a sequential program expecting expressions to return at most one value, web applications ought to cope with user-induced concurrency and be prepared to receive more than one answer to any question: in other words, a call to the `show` function may return more than once, each time within its own new thread.

For the moment assume continuations to be resumed in the store where they were created. Browser-induced concurrency then creates threads within a single store, so managing concurrency is necessary and features such as mutual exclusion, critical sections, semaphores, monitors and so forth are required. Finally the programming language of a web application must take concurrency into account and cannot ignore that it is no longer a pure sequential language. To take this fact into account, new concurrency features were added to Scheme.

The `(fork` π π'`)` special form simply creates two concurrent threads to evaluate the expression π, resp. π' within the current lexical environment and the current continuation. For instance, the evaluation of `(display (fork 1 2))` creates two threads that return independently and concurrently 1 and 2 as argument of the `display` function. The two threads will then print these numbers and continue whatever remains to be done. The `fork` special form is used by the web server to accept connections and service them.

The `suicide` function simply kills the current thread. One use of `suicide` already appeared above in the definition of `service`. This model of concurrency does not offer any "join" operator but a tree of threads where `fork` creates branches while final leaves are terminated with `suicide`.

Threads communicate and synchronize through the shared memory. An atomic swap instruction is provided by the special form `set!` which, atomically, stores a value in a variable and returns its former content. See [31] or [20] for additional semantical detail and examples of higher-level primitives built on top of these features such as `pcall` or `future`.

Our thread model is very simple and ignores the use of `dynamic-wind` recently studied by Gasbischler et al. [12]. According to their taxonomy, our threads are not first-class values but they inherit the entire dynamic environment of their progenitor.

5.1. The `show-once` function

A teacher may want to prevent students from answering the questions of a quiz more than once. This may be achieved if continuations can only be invoked once. In that case, no matter how the "Back" or "Clone" facilities are used, at most one single answer may be expected for any question. The following naïve variant of `show`, named `show-once`, forbids continuations to be re-invoked:

```
(define (show-once page-maker)
  (let ((already? #f))
    (let ((request (show page-maker)))
      (if (set! already? #t)
          (suicide)
          request ) ) ) )
```

The first time the `show` function returns, the `already?` boolean is toggled on and all subsequent calls will commit suicide. This particular definition for the `show-once` function is not very user-friendly since it does not give back any meaningful message to a student that re-answers a question. However, it exhibits how re-invokation may be detected in order to be appropriately handled. More sophisticated uses of web continuations may be devised, see [29, 30] for a library for quizzes where a student may have a "reload" button, may be allowed to answer wrongly at most twice, may be blocked until they provide a good answer, etc.

5.2. Child-less and orphan computations

Since concurrency is present then two new effects may occur: child-less and orphan computations. Usually, for a request, there is at most one response: this is the regular client-server mode. Consider however the following snippet:

```
(let ((request (show some-page-maker)))
 'done )
```

This "child-less" computation ends and does not produce any page to ship as response. This may be easily fixed by wrapping the whole web application within a function (similar to the following `dead-end` function) that immediately responds that there is nothing to say provided, of course, the computation does not invoke `suicide` itself.

```
(define (dead-end anything)
  (dead-end
   (show (lambda (url)
           (html (head (title "END"))
                 (body (p "Nothing to say.")) ) )) ) )
```

Conversely, there may be too many responses as illustrated by the following snippet:

```
(fork (begin (sleep 10) (show some-other-page-maker))
      (show some-page-maker) )
```

In this snippet, the first evaluated `show` expression produces a page and closes the connection towards the client. Ten seconds or so later, a sibling thread produces a second page as result but cannot ship it since the connection is closed and cannot be used again. This thread is said to be an "orphan" since it lost its progenitor. Here in fact, the server wants to reply multiply to a single request of the user (the dual situation motivated the whole paper). At the other end, the browser does not know that there is another response to a previous request since HTTP is a protocol that expects at most one response for any given request.

Since pages should be sent whenever possible to the requester, at least three options seem possible:

- Wait for the end of the computation and accumulate all pages to ship; glue these accumulated pages with some JavaScript code into a single HTTP response and ship it to the client. The glue may, for instance, display the individual pages within separate windows.
- Ship the first synthesized page then accumulate the following pages till a new (and independent) request comes from the client, then ship the accumulated pages glued together as part of the HTTP response of this new request. This scenario, see details in [28], looks like piggybacking the next connection.
- Make the client poll regularly the server in search of additional pages.

The polling scenario is expensive. The piggybacking scenario works only if the client continues to interact with the server; some ergonomy is also needed to not confuse the client who has to cope with past responses mixed with new response.

Waiting for the end of all the computations is clearly not an option since (i) detecting the end is difficult since threads may spawn new threads, threads may signal other threads and induce new computations, continuations may walk again the path of past threads, (ii) to wait for the end precludes interesting effects such as—running a daemon that asks for a remote service whose progress is monitored from time to time or—setting an agent to propose a rendez-vous with another user (in order to start a game for instance), etc., (iii) waiting for a never-ending thread would prevent shipping the pages accumulated so far.

To let the programmer of the web application choose the appropriate behavior is therefore in order. Section 5.3 will show how.

The previous snippet is also an evidence that the body of the `show` function should be in mutual exclusion since it should let only one thread use the current connection before closing it. Replacing `lambda` by `qlambda` (a function whose invokers are queued to ensure mutual exclusion of the body) as proposed by Gabriel and McCarthy [11] should (almost[1]) do the trick.

5.3. Application to top-level

Top-level loops allow programmers to interact with an evaluator. A top-level loop just iterates interactions where an interaction reads an expression, evaluates it and displays its result. Top-level loops usually expect at most one result however adding concurrency to the language allows expressions to return zero, one or many results. A single-programmer interactive Scheme evaluator cannot be naïvely plugged in a multi-user multi-web applications web server. This section describes how the PS3I Scheme evaluator is integrated to the web server.

PS3I is an evaluator packaged as a component. This component is reentrant that is, it may work concurrently for many independent applications and evaluate many expressions without confusion. When a web application is started for a given client, a fresh "world" is created in the evaluator for them. Once this world is created, the web server will systematically associate it to every request coming from this very client to this very web application. A world is made of a store and a global environment (the Scheme global immutable predefined environment). Once created the code of the web application is loaded into that world. Clonable predefined worlds may speed-up these operations.

When a request arrives, the web server creates, in the appropriate world, an *interaction* object in order to control the subtree of threads servicing this request. The web server populates this interaction object with the Scheme expression starting or resuming this web application. Then, the web server triggers it: a thread is created that runs the expression present in the interaction. Threads (recursively) forked by this initial thread will still be sponsored by this interaction object.

An interaction object is equipped with three notifiers: the first notifier (the *value notifier*) is invoked on every produced value, the second notifier (the *exception notifier*) is invoked on every uncaught exception, the third (the *end notifier*) is invoked when the last running thread dies. Additionally, an interaction object may be paused or resumed. When paused, all threads within the interaction object are paused. Resuming the interaction resumes its threads. The interaction is akin to the notion of "group" in the sense of energetic programming, see Moreau and Queinnec [20]. Observe that interaction objects are for the benefit of the web server, they do not need to be available from the web applications though interesting administration web applications might use them.

Observe that the web server does not directly send programs to the Scheme evaluator, instead it creates interaction objects through which the web server triggers and controls, as a whole, the evaluation of all the Scheme expressions these interaction objects contain. This organization allows the web server to decide how to interact with the Scheme evaluator. For instance, the three options presented above for orphan computations are all possible given suitable notifying functions. The end notifier may ship all the pages collected by the value notifier. The value notifier may ship the first page it receives and accumulates the other resulting pages for future use. The value notifier may just accumulate pages in a well-known place in the world; further requests may check this place to discover the newly synthesized pages.

Other behaviors are still possible. The value notifier may pause remaining computations after the first result, the exception may mask anomalies or transform them into pages, the

end notifier may detect that the value notifier was never invoked that is, no page was ever produced, etc.

Instead of an interactive sequential top-level loop asking for programs, a componentized and reentrant Scheme evaluator such as PS3I, (i) is told to prompt for programs (accumulated in an interaction object) then (ii) is told to start (or pause) their evaluation, (iii) reports events.

While this may seem like overkill with respect to the usual request/response behavior, worlds allow a wide variety of behaviors to control the concurrency within the web server. Moreover, since a world contains the store, the environments, the values and the continuations, a world seems an appropriate entity to be made persistent.

6. Representing continuations

For now, continuations were supposed to be held in the memory of the web server that created them and clients know them via their URL. But even with such a simplistic model, one problem is discarding the proliferating unused continuations. The rest of this section discusses this problem.

Responses of the web server(s) include URLs that the clients are free to bookmark, e-mail or memorize in their brain. In all these cases, these URLs are roots for garbage collection and no continuation may ever be discarded. A possible solution used by PS3I, inspired by the behavior of the Session object, is to associate a time-out to any such continuation and to discard it if it is unused for that duration (typically an hour or two). Unfortunately experience [14] suggests that this solution is not entirely satisfactory since it is not possible to foresee the appropriate time-out to use. If too long, the server is cluttered with continuations; if too short, users may lose continuations if they linger to answer (this is particularly problematic for users that are temporarily but involuntarily disconnected from the Internet).

A radically different solution, first advocated by Hughes [16] then adopted by Thiemann [32] and Graunke et al. [19], is to store continuations on clients. The continuation is serialized and stored in the page shipped to the client in some hidden fields or cookies. This solution has an obvious advantage: the server is entirely free of continuations, it does not have to record them, to discard them, to manage them. The server is now stateless (as HTTP) and when a client sends a request, the request contains the continuation to resume.

6.1. Serializing continuations

Continuations must be serialized in order to be shipped to clients. Continuations must also be serialized if the web server must be persistent—that is, must be resistant to shutdown or failures. In this case, continuations are stored within some files, RDBMS or LDAP data stores. Serializing continuations in such places also allows multiple servers to be clustered in order to support the load since continuations may now migrate from one server to another.

Serializing continuations *per se* depends principally on the way the implementation represents the control stack. To serialize a continuation requires also to serialize the environments it closes as well as all other reachable values. However, it is difficult to distinguish, at serialization-time, between the continuation and the store. Thus, serializing a continuation nearly amounts to serialize the entire world that contains it. A world comprises

immutable parts such as the Scheme global immutable predefined environment, the code of the web application and so forth. These immutable parts do not require to be serialized in their finest details since they may be easily regenerated. Additionally, as already noted by Danvy [10], most continuations share a lot and so do web continuations, and serialization may take advantage of that fact.

As usual with serialization, there exist unserializable values such as open network connections, mutexes and other resources managed by the operating system such as threads. All these values should be avoided or rebuilt at deserialization-time (files are for instance reopened and so are connections to databases). This implies some sort of protocol to specify how values may be properly deserialized.

To avoid serializing running threads, PS3I serializes continuations and worlds independently. Continuations are just registered and memorized in their current world. A world is serialized only after the last thread of the last interaction dies (a specific notifier detects that situation). Therefore a world is either active in memory or, stable in some persistent storage. Before a continuation is resumed, the appropriate world is determined and reinstalled from persistent storage if necessary.

From my experience, the control stack of a web application is usually very shallow. For instance, there is only one type of continuation within the Sum Web Application, it mainly amounts to two pending `let` blocks that is, `(let ((next (let ((request ...` and two captured variables: `numbers-so-far` (the list of previously input numbers) and `page-maker` (the closure that displays an addition which itself also captures `numbers-so-far`).

6.2. Client-side continuations

When a continuation and its accompanying store is serialized and shipped to a client, one problem is how to deserialize it while complying with Scheme semantics. For instance, using `show-once` toggles a local mutable variable. This mutable variable should then be toggled on for every (serialized) continuation having captured it. If a continuation and its associated store is deserialized on two different servers then the mutable variable is duplicated and toggling one of its instance does not toggle the other instances: this no longer respects Scheme semantics.

In order to have a stateless server and to record continuations on clients, one solution is to completely avoid shared mutable variables as in Haskell. Alas, this prevents defining functions such as `show-once` and answering twice to a question will lead to two different threads in two different stores (though they can still exchange information and synchronize via the file systems, databases, etc. see Thiemann's WASH/CGI system [32]).

A very ingenious solution, advocated by Graunke et al. [14], is to record stores within cookies while continuations may still be recorded within hidden fields. Hidden fields are returned to the server when a form containing them is submitted to the server. On the other hand, cookies are returned to the server whenever any form is submitted to the server. When a server updates a cookie, all future forms will normally be submitted with that fresh cookie. Whenever the store is updated and sent to a browser as a cookie, this cookie will update previous versions of this very store in that browser.

Alas, the latency of the network varies, browsers are multi-threaded and, quite often, browsers do not return their freshest cookies. The latest point may be partially fixed by a new RFC enhancing the semantics of cookies to incorporate versioning: browsers will then determine easily the most up-to-date value to send for a cookie. But even if cookies are up-to-date when sent by browsers, this does not ensure that they are still up-to-date when received by web servers. So, another solution proposed by Graunke et al. [14] labels cookies with a version number and requires, on the server-side, to memorize, for any cookie, the last version sent. This requires some memorization on the server-side in order to be able to reject all but the last version of the cookie. This memorization is quite small since only the version number has to be remembered for any user running any web application.

Shipping continuations and their stores into clients freezes the store. Threads should not run on the server after this freezing since, in absence of further requests coming from the client, the resulting store would not be saved. Shipping continuations also makes it harder to design multi-user web applications where several clients share mutable data and exchange continuations (as in a chess game where players may wish to permute white and black positions).

Another related problem with continuations stored on the client-side is that clients may alter them thus making the information they contain totally insecure. However this problem is easily solved if continuations are encrypted.

6.3. Comparison

Both solutions offer advantages and inconvenience summarized below.

Storing continuations on the server side naturally copes with Scheme semantics (provided worlds are not duplicated). It also allows to share parts of worlds or to let threads run after shipping responses. Furthermore, worlds may be inspected so I can check, on the server, whether my students did their assignments. Continuations are discarded after some blind time-out.

Storing continuations on the client side requires higher bandwidth, freezes the store at serialization-time and imposes some sort of synchronization with external data store (file, DB) to cope with mutable data structure if needed. On the bright side, servers have a lighter management of continuations.

7. Scope

In Section 3 appeared the `current-connection` function. This section investigates how to implement that function and proposes a new concept of scope.

To ship a page towards a client requires knowing which connection to write on. When a connection is accepted by the listening web server, this connection must be preserved somewhere in order to be retrieved by `current-connection`. The most obvious solution would be to place this connection in the HTTP request object returned as the value of `show`. Alas, this would impose the web application to pass the request object to every function that {calls a function that}*may eventually call `show`. This will amount to a sort

of "connection-passing style", a mere annoyance and a source of obfuscation destroying direct style.

To store the request in a shared global variable is not an option because of concurrency. Local variables are not a solution either since, most often, there is no local (neither lexical nor dynamic, also known as "fluid" or "parameter" in some dialects of Scheme) environment that embeds two consecutive calls to `show`. To record the connection in a thread-local storage [12] is not either a solution since this storage would not be passed onto `fork`-ed threads.

Actually, what is needed is to ensure that this very connection is the result of `(current-connection)` anywhere in and anytime during the computation resumed by `(k request)` within the definition of the `service` function. Similarly to the `java.lang.InheritableThreadLocal` concept of Core Java, our system proposes to attach information to the thread and to all of its offspring: this new scope is named the "thread + offspring" scope. Two primitives are offered to manage that scope that is, one primitive to put (or update) information in this scope under a given name and another one to get the associated information. These are:

```
(thread+offspring-put' name' value)
(thread+offspring-get' name' value-if-absent)
```

With these primitives, the `service` and `current-connection` functions become:

```
; HTTPRequest → Nothing
(define (service request)
  (let ((k (get-registered-continuation
            (get-request-url request) )))
    (if k
        (begin
          (thread+offspring-put' connection
                                 (get-request-connection request) )
          (k request) )
        (error "Missing continuation") ) ) )

; Nothing → OutputPort
(define (current-connection)
  (let ((connection (thread+offspring-get' connection #f)))
    (or connection
        (error "No connection!") ) ) )
```

Observe, in the `service` function, that the connection is put into the thread + offspring scope before the continuation is resumed. When the continuation is resumed, the lexical and dynamic scope of the current thread are reset to what they were when the continuation was captured. This proves that the connection cannot be held in these environments.

Moreover, not only does the thread + offspring scope accompany the current thread, it is also propagated to the offspring of the current thread. This is needed since, otherwise, forked threads will not have access to the current connection (as shown in the snippet

describing the orphan problem). The model of threads proposed in this paper corresponds to a tree where branches are created with `fork` while leaves are identified by calls to `suicide`. The thread + offspring scope exactly corresponds to a subtree. This model is orthogonal to continuations and (lexical or dynamic) environments.

JavaServer Pages [23] (or JSP for short) proposes a number of specific scopes. The notion of thread + offspring scope is somewhat intermediate between the "request scope" and the "session scope" of JSP. It contains request scope since it covers all the threads that participate to the response of a request but JSP limits the use of that information to the first response while the thread + offspring scope makes that information available until the last thread of the offspring commits `suicide`. The thread + offspring scope is contained within the session scope since this one embeds the whole web application.

8. Related work

Events can be processed by invoking continuations as shown by Clinger et al. [7, 8]. This paper furthers this idea for web interactions.

Zellweger [34] proposed "scripted paths" with sequences, conditions, procedures and parallelism. In her realization, the Scripted Document system, trails were represented by a list of pairs made of a page and an action (a piece of code). The action may for instance scan the page to extract voice annotations, play them and switch automatically to the following page of the trail. Our solution improves on that view since our trails are real programs, more agile and not page-centric.

Walden's paths [2] are sequential trails over already existing pages, our model adds a programmable view over these trails allowing the rest of the trail to be computed and thus to depend on information gleaned from previous pages. We may even embed a trail (for instance, a trail built around a quiz) as a sub-trail of a wider trail.

MAWL [18] standing for—The Mother of All Web Languages—is an application language for programming interactive services in the context of the World Wide Web. My system shares a lot with MAWL: pages are viewed as functions and continuation URLs are synthesized. However, it differs from MAWL on a number of points: continuations are handled explicitly (and this allows to build more sophisticated interactions), it uses an existing language, Scheme, rather than a specialized language (but it lacks static analyses such as type checking, see however [19]).

<`bigwig`> is a conceptual descendant of MAWL [4] and is itself superseded by JWIG, a new Java-based enhanced system [6]. The runtime of <`bigwig`> and JWIG uses persistent threads for web applications, such a persistent thread materializes a place holder for the latest continuation only. When returning to a previous continuation, the web dialog is restarted afresh.

Touchette [33] addresses the problem of the non-repeatability of transactions in presence of the "Back" and "Clone" buttons. Provided you master continuations, I believe direct style and `show-once` to be more robust, more systematic and more customizable.

Monads are strongly related to continuations; Hughes generalized monads into arrows [16]. One application is the encoding of the continuation of a web application written with arrows, as an URL (and some hidden fields) sent to the client. This was further developed by

Thiemann and his WASH/CGI system [32] accompanied by many useful features (typing, XML validation, well defined sub-languages, etc.) This solves the problem of multiple submissions from the same form since submissions bring their accompanying state. It solves also very nicely the garbage collection problem of the server since the server does not need to store these continuations: they're recorded within clients' pages. While my scheme is rather independent[2] on which side records continuations, I tend to record them in the server as parts of the state of the Scheme evaluator which I may examine to check whether students accomplish their assignment.

This paper (and especially Section 6) would have been a pale rephrasing of the original paper [28] without the series of papers by Graunke et al., see [14, 15, 19]. They explore a variety of aspects, discover the problem of the store and solve it with cookies. I also used their Scheme system, DrScheme, to build a new educational CD-ROM whose topic is the Scheme programming language [5] with the techniques originally implemented with PS3I. This was easier than in PS3I since there was only one user and one sequential web application.

The fact that a web application is a single program also leads to an exciting idea from Perugini and Ramakrishnan [24]. Out-of-turn information (where the user sends information not asked for by the server) allows the server to partially evaluate the continuation thus providing a kind of customization of the web application.

9. Conclusions

This paper discusses the use of continuations for web applications. Continuations allow framework designers to build libraries defining typical uses of continuations. Programmers may then use these libraries to write web applications in direct style. However, if not hidden by libraries, programmers must take into account that any question addressed to the user may return more than one answer (a normal fact of Web life).

An additional benefit, for teachers, makes web programming a very gentle and adequate introduction to continuations and CPS (Continuation Passing Style).

Acknowledgment

Thanks to David De Roure who mentioned the word "continuation" when talking of browsers circa 1995. Great thanks for the referees who incredibly improved that paper.

Notes

1. `qlambda` was invented for a Lisp that lacks continuations with indefinite extent. `qlambda` for Scheme should prevent a programmer to capture a continuation within the body of a `qlambda` and invoke it, otherwise this programmer may experiment more than one thread within the same mutually exclusive body.
2. To achieve that effect, just change, in the definition of `show` and `service`, the call to `register-continuation!` to some serializing function and the call to `get-registered-continuation` to some unserializing function.

References

1. Cocoon developers. The Apache Cocoon Project. http://cocoon.apache.org/
2. Shipman III, F.M., Marshall, C.C., Furuta, R., Brenner, D.A., Hsieh, H.-W., and Kumar, V. Creating educational guided paths over the world-wide web. In *Proceedings of Ed-Telecom '96, Association for the Advancement of Computers in Education*. Boston: Massachusetts, USA, 1996, pp. 326–331.
3. The Apache Jakarta Project. Tomcat. http://jakarta.apache.org/
4. Braband, C., Møller, A., Sandholm, A., and Schwartzbach, M.I. A runtime system for interactive web services. *Journal of Computer Networks*, **31**(11–16) (1999) 1391–1401.
5. Brygoo, A., Durand, T., Manoury, P., Queinnec, C., and Soria, M. Experiment around a training engine. In *IFIP WCC 2002–World Computer Congress*. IFIP. Montréal Canada, Aug. 2002.
6. Christensen, A.S., Møller, A., and Schwartzbach, M.I. Extending Java for high-level web service construction. *ACM Transaction on Programming Languages and Systems*, **25** (2003) 814–875.
7. Clinger, W.D., Hartheimer, A., and Ost, E. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, **12**(1) (1999) 7–45.
8. Clinger, W.D., Hartheimer, A.H., and Ost, E.M. Implementation strategies for continuations. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, Aug. 1988, pp. 124–131.
9. Coward, D. *Java*™ *Servlet Specification, v2.3 pre-FCS*. SUN Microsystems, Aug. 2000.
10. Danvy, O. Memory allocation and higher-order functions. In *PLDI '87—ACM SIGPLAN Programming Languages Design and Implementation*, 1987, pp. 241–252.
11. Gabriel, R.P. and McCarthy, J. Queue-based multi-processing Lisp. In *LFP '84—ACM Symposium on Lisp and Functional Programming*, 1984, pp. 25–45.
12. Gasbischler, M., Knauel, E., Sperber, M., and Kelsey, R.A. How to add threads to a sequential language without getting tangled up. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*. Boston: Massachusetts, USA, Nov. 2003.
13. Graham, P. Beating the averages. http://www.paulgraham.com/avg.html
14. Graunke, P.T., Findler, R.B., Krishnamurthi, S., and Felleisen, M. Automatically restructuring programs for the web. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001)*. M.S. Feather and M. Goedicke (Eds.). IEEE Computer Society: Coronado Island, San Diego, California, USA, Nov. 2001, pp. 211–222.
15. Graunke, P.T., Krishnamurthi, S., Van Der Hoeven, S., and Felleisen, M. Programming the web with high-level programming languages. In *Proceedings of the Tenth European Symposium on Programming (ESOP 2001)*. D. Sands (Ed.). Springer-Verlag: Genova Italy, April 2001, pp. 122–136.
16. Hughes, J. Generalising monads to arrows. *Science of Computer Programming*, **37**(1–3) (2000) 67–111.
17. Kelsey, R., Clinger, W., and Rees, J. (Eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, **11**(1) (1998) 7–105. Also appears in *ACM SIGPLAN Notices*, **33**(9) (1998).
18. Ladd, D. and Ramming, J. Programming the web: An application-oriented language for hypermedia services. In *4th International World Wide Web Conference—WWW4, World Wide Web Consortium*. Boston: Massachusetts, USA, Dec. 1995, pp. 567–586.
19. Matthews, J., Findler, R.B., Graunke, P., Krishnamurthi, S., and Felleisen, M. Automatically restructuring software for the web. *Journal of Automated Software Engineering*, (2004).
20. Moreau, L. and Queinnec, C. Design and semantics of quantum: A language to control resource consumption in distributed computing. In *Usenix Conference on Domain Specific Language, DSL'97*. Santa-Barbara: California, USA, Oct. 1997, pp. 183–197.
21. Nørmark, K. Using Lisp as a markup language—The LAML approach. In *European Lisp User Group Meeting*: Amsterdam: Holland, 1999.
22. Open Market, inc. Fastcgi: A high-performance web server interface. Technical White Paper, April 1996. http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm
23. Pelegrí-Llopart, E. and Cable, L. *JavaServer Pages*™ *Specification, version 1.1*. SUN Microsystems, Nov. 1999.
24. Perugini, S. and Ramakrishnan, N. Personalizing web sites with mixed-initiative interaction. *IEEE IT Professional*, **5**(2) (2003).

25. Predescu, O. Model-view-controller in Cocoon using continuations-based control flow. Ovidiu Predescu's Weblog (Sept. 2002). http://www.webweavertech.com/ovidiu/weblog.
26. Queinnec, C. *Lisp in Small Pieces*. Cambridge University Press, 1996. Paperback reprint 2003.
27. Queinnec, C. Enseignement du langage C á l'aide d'un c'ed'erom et d'un site—Architecture logicielle. In *Colloque international—Technologie de l'Information et de la Communication dans les Enseignements d'ing'enieurs et dans l'industrie—TICE 2000*. CNED, Troyes: France, Oct. 2000, pp. 93–102.
28. Queinnec, C. The influence of browsers on evaluators or, continuations to program web servers. In *ICFP '2000—International Conference on Functional Programming*: Montréal: Canada, Sept. 2000, pp. 23–33.
29. Queinnec, C. A library for quizzes. In *Scheme 2002—Proceedings of the Third Workshop on Scheme and Functional Programming*. S. Olin (Ed.). Pittsburgh: Pennsylvania, USA, Oct. 2002, pp. 1–7.
30. Queinnec, C. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Notices*, **38**(2) (2003) 57–64.
31. Queinnec, C. and De Roure, D. Design of a concurrent and distributed language. In *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*. R.H. Halstead Jr and T. Ito (Eds.). Boston: Massachusetts, USA, Oct. 1993, pp. 234–259.
32. Thiemann, P. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *PADL'02—Practical Applications of Declarative Languages*. Lecture Notes in Computer Science. Portland Oregon, USA, Jan. 2002, pp. 192–208.
33. Touchette, J.-F. HTML thin client and transactions. *Dr. Dobb's Journal, Software Tools for the Professional Programmer*, **24**(10) (1999) 80–86.
34. Zellweger, P.T. Scripted documents: A hypermedia path mechanism. In *Proceedings of Hypertext-89*. Pittsburgh: Pennsylvania, USA, 1989, pp. 1–14.