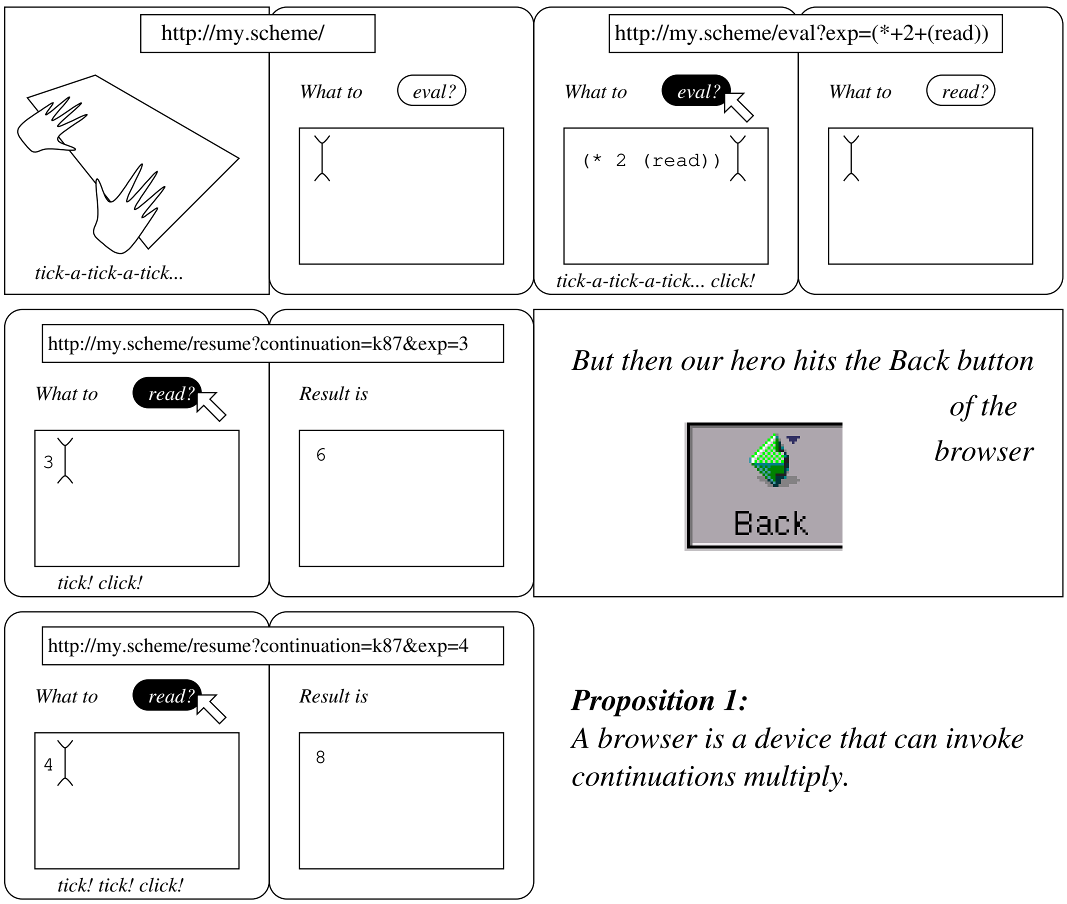
The Influence of Browsers on Evaluators or, Continuations to Program Web Servers (revised 3 IX 2000)

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



Proposition 1:
A browser is a device that can invoke continuations multiply.

ABSTRACT

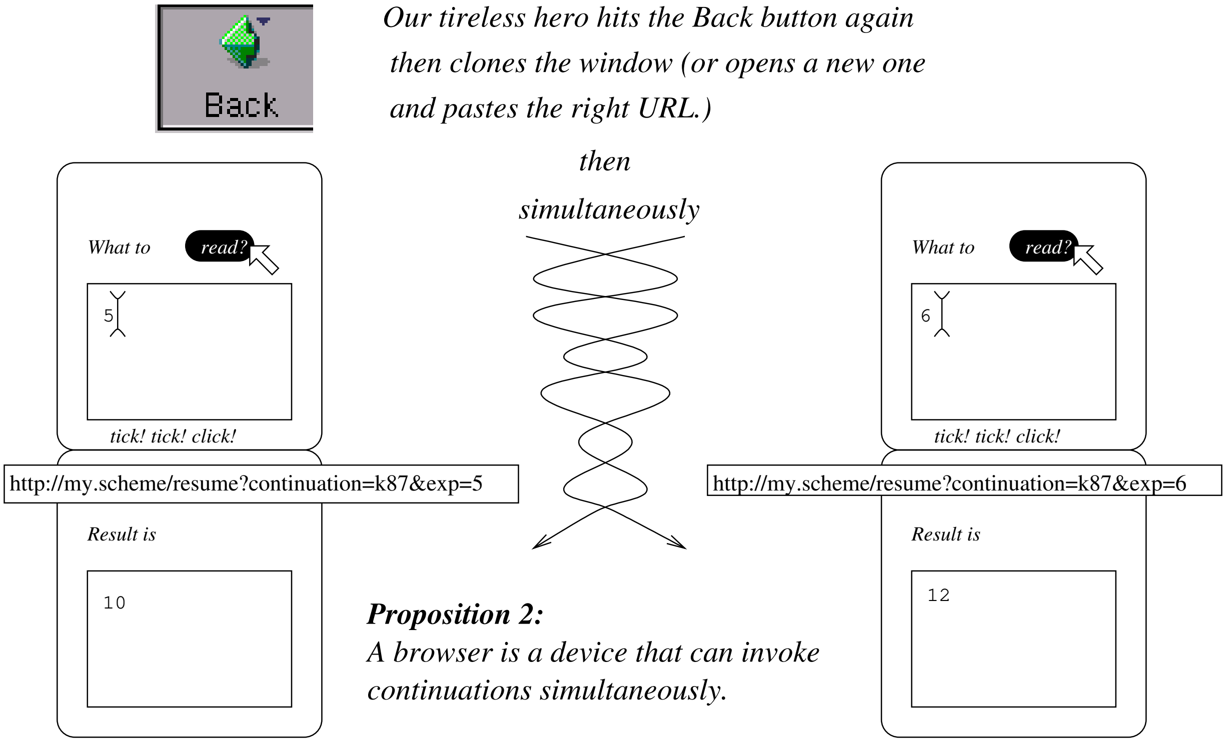
While developing the software of a browser-operated educational CD-ROM, we had to face a number of problems. This paper presents these problems and the solutions we found. Amusingly, most of our solutions rely on continuations. Are browsers and multimedia the future of continuations ?

ICFP '00, Montréal, Canada.

Through their "Back" button or "Clone window" menu item, browsers have powerful abilities that force servers to take care of multiply and simultaneously answered questions. A comprehensive tool to apprehend these problems as well as to solve them is to view these abilities as operators acting on the continuations of the computation performed by servers.

Thematical trails are provided to walk through the CD-ROM but do not prevent students to wander elsewhere. A trail may contain choices or quizzes so the rest of the trail should adapt to the walked part. We consider the trail as a computation and the position of the student as a continuation within that computation.

Moreover this paper advocates a computation-centric view of servers (in opposition to the usual page-centric view) where interactions with users suspend the computation into



continuations that may be later resumed. This approach is superior because the continuation reifies, automatically and without errors, the whole state of the computation.

Keywords

Scheme, continuations, hypertext, WWW

1. MOTIVATION

Browsers are ubiquitous. They tend to replace traditional window systems because, like them, they offer menus, buttons and text rendering but, unlike them, they are everywhere and (nearly) the same throughout the world. Consequently more and more computations are driven by browsers be it to set parameters, to monitor progress or to display results.

However browsers have two important capabilities. First, they maintain a set of bookmarks as well as an history of visited URLs or, more precisely, a history of HTTP requests and their associated resulting page. Via the "Back" and "Forward" buttons, the user may come back and forth through this history and browse the past visited pages. This leads to a "backtracking" style of surfing when the user goes back to some old page and tries another path through the web maze.

Second, browsers allow users to follow a link while opening a new window making easier to explore more than one trail concurrently. This may be considered as the creation of a new thread in user's brain. This thread ends when the window is closed.

These two familiar capabilities interact weirdly with web computations i.e., computations involving the user through more than one page to achieve some goal. With the "Back" button, a user may come back to a page (containing a question from the server to the browser) and submit another answer to an already answered question, see first strip. Of course, the new answer is probably different from the past answer and this is a perfectly understandable behavior since *(i)* a user may fix a previous typo that was only perceptible a few pages later, or *(ii)* a gamer may withdraw the last turn, or *(iii)* a user may be interested in a kind of "what-if" computation i.e., setting a parameter, looking for its effect and coming back to tune it more finely.

The "Back" button is a local amenity of the browser but servers have no clue about its use. The HTTP protocol is state-less and a server just sees incoming requests. Thanks to cookies, hidden fields or URL-rewriting, servers may discriminate browsers but still cannot perceive the use of the "Back" or "Forward" buttons, nor distinguish between different windows within a same browser.

Xmosaic from NCSA used to provide a menu item to clone the current window (as well as its history). Although this facility seems to have disappeared in Communicator and Explorer, one can roughly approximate it by opening a new window and pasting the current URL within it or, by bookmarking the current URL, opening a new window and asking it to fetch the freshly bookmarked URL. Suppose that a single page is displayed in two different windows, suppose that that page contains a question then, the user may submit two different answers concurrently to the same question (see second strip): this only requires the user to have some mouse-moving and mouse-clicking proficiency.

In most servers, incoming requests are handled by different threads. Therefore the two previous, concurrently sent, requests most naturally lead to two concurrent threads in the server. These threads start in the same state: the one that corresponds to the original question. Servers are often puzzled by requests answering an already answered question

[18], they sink into a lot of troubles when two answers come concurrently for the same question!

This paper argues that continuations are the appropriate tool to apprehend these problems. When a server produces a page to be displayed in a browser, it suspends its computation while allowing its resumption later. The server reifies the "continuation of the page" and associates it to some URL (probably appearing as a link somewhere in the HTML page itself). When a request is sent to that very URL, the continuation is resumed (possibly with some fresh values to influence the rest of the computation).

The continuation of the page is the continuation of the web computation that page is involved in. No matter how complex the computation is, continuations are well defined everywhere. Continuations deliver the coder of the web computation from designing only small, simple, page-centric automata which are characterized by a simple state and a small matrix of transitions triggered by requests. Continuations automatically capture the computation state and this greatly frees the coder.

Actually all these ideas sprung from the design of an educational CD-ROM for which we realized a persistent server of pages implementing some thematical trails. A trail is a way to traverse a subset of the pages of the CD-ROM. The trail should be reactive so it may adapt to the student, it should be persistent since it may be followed for months, it should be independent of the traversed pages. It took time for us to discover that these trails were in fact programs and, a student's position within the trail: a continuation!

To sum up, the use of a browser to compute over the web naturally requires the language, in which computations are expressed, to support continuations and concurrency. The rest of the paper explores this domain at the cross of continuation, hypertext and web [1]. For ease of exposition, we will use the Scheme programming language [8].

In Section 2, we will analyze the linguistic features associated to the browser- and the server-sides and their relationship. We will give an example of a web computation illustrating continuations. This example is similar to one we used in lecture to motivate students to grasp what are continuations.

However continuations and concurrency introduce new problems as shown in Section 3: computations may yield no result or more than one result. We will expose our solutions and introduce a new kind of scope: the thread+offspring scope.

Finally, we will present two applications putting these ideas to work in Section 4 — a multi-user, multi-thread, browser-operated Scheme interpreter and — some fine points of the CD-ROM itself and its trails.

Related works and conclusions end the paper.

2. FEATURES

We present in this Section the linguistic aspects related to the browser-side and the server-side. A first example of a web computation is discussed.

2.1 Browser-side

There are three actions a user may perform on a browser:

1. A user may `submit` information to a server via a GET or POST request (the main types of request of the HTTP protocol). The request opens a connection to the server through which the browser will receive back a page to display.

2. A user may asynchronously `stop` a request to signify that the page is not desired any longer. This may be expressed by hitting the "Stop" button or by closing the window from where the request was submitted. Stopping the request closes definitely the connection.

3. A user may `clone` a window and initiate a new independent and concurrent activity in the new window.

Note that these actions may also be performed from a command line interpreter from which a user may submit requests by hand (telnet, wget, etc.) or by script, sequentially or concurrently. There, the user also has the power to interrupt submissions but is lacking the HTML rendering ability.

Observe that there are three entities: windows, connections and requests with different lifetime. A window may host, one at a time, a number of pages. At any moment, a window is associated to at most one open connection[1]. A connection is closed when stopped or when the resulting page is served; once closed, it remains definitely closed. A connection carries a request from the browser to the server and will convey the resulting page from the server to the browser. Requests are memorized by the browser and may be submitted again (with the "Reload" button) thus forcing the creation of a new connection.

2.2 Server-side

We assume that whenever a connection is accepted, a request is extracted from it and a thread is created to service that request. Some HTML is output to the connection while the connection is open. When the request finishes, the thread flushes the output, closes the connection and terminates. The browser has no knowledge of user's windows and only sees connections and requests.

Our model of concurrency is very simple.

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, while evaluating `(display (fork 1 2))`, two threads are created that return independently and concurrently 1 and 2 as argument of the `display` function. The two threads will then print these numbers and continue what remains to be done.

The `(suicide)` function simply kills the current thread.

Our model of concurrency does not offer any "join" operator but a tree of threads where nodes are created by `fork` and final leaves are terminated with `suicide`.

Threads communicate and synchronize through 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 [15] or [10] for additional semantical details and examples of higher-level primitives built on top of these features such as `pcall` or `future`.

A web computation is represented by a program evaluated by the server. In this program, we express that a page should be served to a browser with the `show` function. The `show` function takes a page as argument, captures its continuation, registers that continuation under a fresh and public URL and, finally, asks the page to produce some HTML,

[1] We neglect frames or images pre-fetching.

closes the connection and kills the current thread. The page is represented by a function that expects a single argument: the URL of the continuation. The following definition embodies that behavior:

```
(define (show page)
  (call/cc
   (lambda (resume)
     (let ((url (register-continuation! resume))
           (connection (current-connection)) )
       (display (page url) connection)
       (close-output-port connection)
       (suicide) ) ) ) )
```

When a browser resumes a web continuation, it submits a regular request with some information. This information corresponds to the filled text fields and/or selected choices of the forms within the currently displayed page. The requested URL may look like the ones used in the strips, for instance:

`http://my.scheme/resume?continuation=k87&exp=4`

Given a resumption URL (i.e., an URL named `resume`), the server creates a thread, finds the registered continuation (whose name here is `k87`) and resumes that continuation with the request object. The server looks like the next function though it does not need to be written in Scheme:

```
(define (server request)
  (let* ((c (get-parameter request "continuation"))
         (k (get-registered-continuation c)) )
    (if k
        (k request)
        ... ) ) )
```

The request object may be seen as the union of the `httpRequest` and `httpResponse` objects in Servlets parlance [5]. It contains parameters i.e., name-value pairs such as `exp` and `4` (in the URL, they start with a `?` and are separated with `&`). We will use the `get-parameter` function to access the value of the parameters.

From the web computation point of view, it looks as if the `show` function ships out a page, blocks until a request for its continuation arrives and returns as result such a request object. We emphasize that no thread is blocked (since the mere existence of such a thread would not allow the web continuation to be resumed more than once), the continuation of the page is only reified and recorded. When an appropriate request arrives, the server creates a new thread to resume that named continuation. This thread will die at the next call to `show` (or, more directly, by `suicide` of course).

2.3 Web computation

Let us give a very simple, but complete, example of a web computation. The next file holds a complete web computation and includes three pages as well as the transition logic between these pages.

When started, this computation asks for a conversion rate between French Francs (FRF) and another currency, then asks for an amount of Francs to convert and, finally, displays the result of the conversion, see Figure 1. The two pages asking for numbers have a “Continue” button to resume the conversion process i.e., the continuation.

```
(define (conversion)
  (let* ((req1 (show (mk-read-rate-page)))
         (rate (string->number
                (get-parameter req1 "rate") ))
         (currency (get-parameter
                    req1 "currency" )) )
    (if (and rate (> rate 0.0))
        (let* ((req2 (show (mk-read-francs-page
                            currency )))
               (francs (string->number
                        (get-parameter
                         req2 "francs" ) )) )
          (show (mk-result-page rate
                                francs
                                currency )) )
        ;;incorrect rate
        (conversion) ) ) )
(define (mk-read-rate-page)
  (lambda (kurl)
    (html (head (title "Conversion"))
     (body (form method: 'post action: kurl
            (p "rate "
               (input type: 'text size: 10
                      name: 'rate ) )
            (p "currency "
               (input type: 'text size: 10
                      name: 'currency ) )
            (input type: 'submit
                   value: "Continue" ) )) ) ) )
(define (mk-read-francs-page currency)
  (lambda (kurl)
    (html (head (title "How many Francs?"))
     (body (form method: 'post action: kurl
            (p "Converting into " currency)
            (p "Francs "
               (input type: 'text size: 10
                      name: 'francs )
            (input type: 'submit
                   value: "Continue" ) )) ) ) ) )
(define (mk-result-page rate francs currency)
  (lambda (ignored-continuation-url)
    (html (head (title "Conversion result"))
          (body (p "If 1 FRF corresponds to " rate
                   " " currency " then " francs
                   " FRF correspond to "
                   (* rate francs) " " currency
                   "." )) ) ) )
(conversion)
```

The `conversion` function clearly and totally expresses the dependences between the three pages of the web computation. These pages are created with the three functions whose name is prefixed by `mk.`. As before, pages are unary functions expecting the URL of their continuation and returning the HTML of the page. We borrowed the very elegant solution proposed by Nørmark, which he named LAML standing for Lisp Abstracted Markup Language [11][2]. With his technique, HTML tags (`html`, `p`, `input` ...) are emitted via a library of functions; tags attributes (`type:`, `method:`) are specified via keywords to these functions.

Observe in this program that we verify that the rate obtained by the first page `read-rate-page` is indeed a valid number. If this is not the case then the `conversion` function is called again with the same continuation. We do not

[2]Additional details about LAML may be found on Nørmark's site at `http://www.cs.auc.dk/~normark/`.

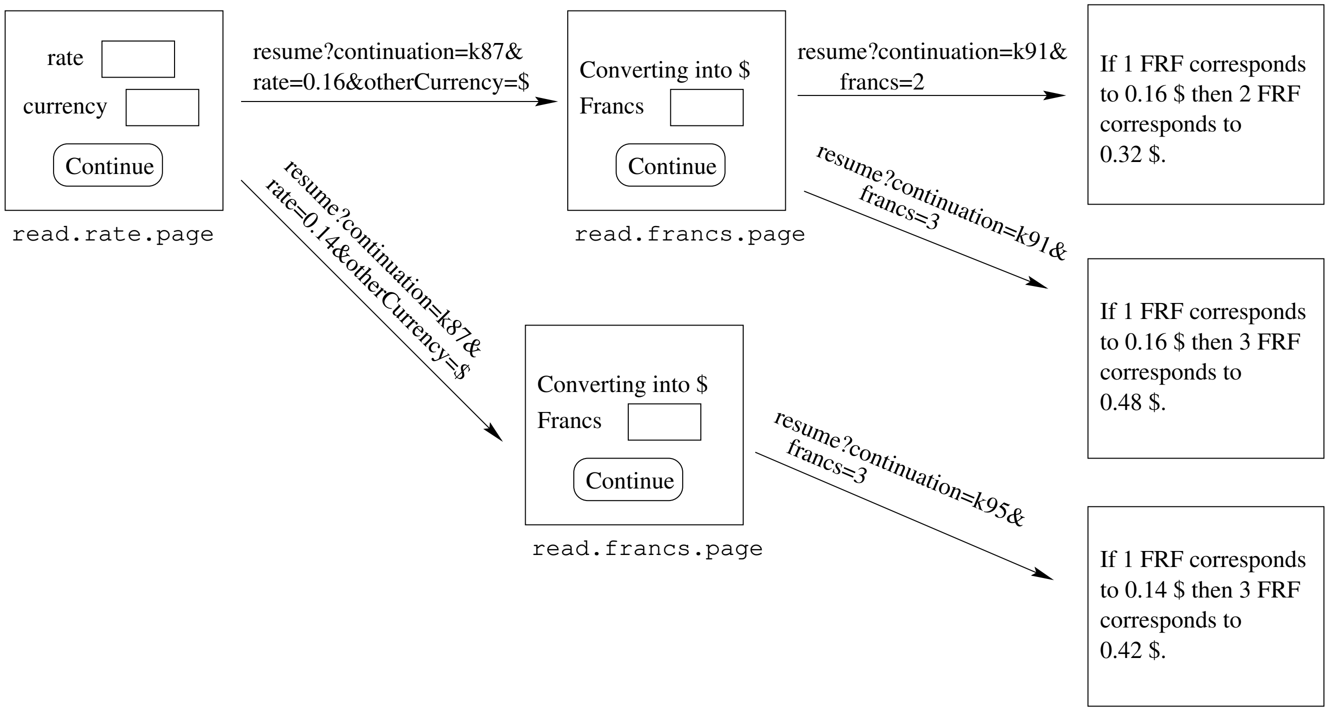


Figure 1: The "conversion" web computation

check other things to leave the definition of `conversion` unencumbered.

When the second page `read-francs-page` is displayed, the continuation already embeds the rate to apply as well as the name of the other currency. Thanks to continuations, after obtaining the final page, the user may go back to the previous `read-francs-page` page and ask to convert another amount of Francs, see Figure 1. The user may also go back and back to the `read-rate-page` page and alter it to compare, see Figure 1 again. The user may also clone the second resulting page, go back, change the rate and, finally, have two windows converting Francs with two different rates. Note that while these two pages look the same and are based on the same `read-francs-page` template, they are different because their continuations are different, because they embed different rates and their "Continue" button resume different computations.

Since pages are generated via function calls, they take their arguments by regular invocation rather than by global mutable shared variables. To be independent of side-effect strengthens this code in presence of concurrency.

All these operations can be done since all displayed pages have a well defined web continuation encapsulating what remains to be done according to the web program. Continuations allow the coder to shift from a page-centric view to a computation-centric view and use regular linguistic features such as scope to manage intermediate data.

2.4 Analogies

It is necessary that the language of web computations offers the `fork` special form since the `clone` facility on the browser-side allows to simulate it.

The HTTP protocol offers a number of features among which is redirection. With redirection, a server may answer a browser to suggest it to re-send its original request towards another URL. This is mainly used to cope with pages or sites that changed their location.

With redirection and JavaScript-ing ability, it is a simple matter to simulate `fork` even if absent. A form such as (`fork` π π') may be implemented as:

```
(if (equal? (get-parameter (show fork-page) "exp")
            "t" )
    π
    π' )
```

The `fork-page` is defined with a small snippet of JavaScript.

```
(define fork-page
  (lambda (kUrl)
    (html (script language: "JavaScript" "
  var kurl = \"http://my.scheme/resume\"
      + \"?continuation=\" + \"" kUrl "\";
  window.open(\"" kurl "\" + \"&exp=t\",
              \"UselessWindow\",
              \"copyhistory\");
  window.location.href = \"" kurl
     "\" + \"&exp=f\"; " )) ) )
```

This snippet opens another browser window[3] that immediately requests the continuation to be resumed with the `"t"` value[4]. Meanwhile, the current window also requests the same continuation to be resumed with the `"f"` value. We therefore obtain two threads evaluating π and π' con-

[3] The `"copyhistory"` option is useless here but shows how a window may be programmatically cloned as in Xmosaic.

[4] We do not use the regular true value of Scheme (expressed as `#t`) since the hash sign has another interpretation in URLs!

currently so we simulate the `fork` intention at the price of a new window in the browser[5].

There exists a weaker analogy between `suicide` and `stop`. They are less tightly coupled since `suicide` immediately kills the current thread (on the server-side) while `stop` immediately closes the connection (on the browser-side) and thus allows the thread, within the server, to linger until it starts to produce some output and be notified of an IO error on the closed connection.

2.5 Pedagogy

Students are very used to surf the web. This year, we introduced continuations through the example of the first strip and found that our students had no reluctance to consider and use continuations provided they were named by URLs. After some examples of web computations akin to the previous one, they grasped the concept of continuations and it was time to deliver more classical lectures on CPS (Continuation Passing Style) *i tutti quanti*.

That continuations and web are entangled is rather exciting for students especially when they realize that they may suspend quite complex computations and resume them with help of continuations without having to code a state stuffed with all the data needed to resume it. Continuations automatically capture what is needed to resume computation. We recommend this pedagogical approach!

3. PROBLEMS

The "Back" and "Forward" buttons are often disconsidered from the user's point of view since *(i)* most of the time these buttons lead to so called "landmark" pages that are already reachable through flashy icon-highlighted links within the displayed page; for instance, the home-page, the site-map, the FAQ page are generally referred to from the header and/or the footer of every page of a site; *(ii)* caching strategies at various levels lessen the cost of re-asking for a page rather than going back locally to see it again.

From the programmer's point of view, to make use of these buttons to invoke multiply and even concurrently continuations poses a number of problems to servers [18]. In regular languages, an expression has at most one value: most often one and zero in case of exception. In languages with first-class continuations, an expression may have an undefined number of values since its continuation may be invoked multiply (take care that we are not speaking of an expression that returns a single "multiple value" result (as can be done in Scheme R5RS with `values` and `call-with-values`), we are speaking of an expression that returns a single value every time it returns). Corollarily, it is difficult to determine the end of a computation that is, the moment when an expression is guaranteed to return no more values. For instance, this makes critical section a somewhat inappropriate concept since if one thread enters it, captures a continuation and makes it visible, an undefined number of threads may exit the critical section just using the available continuation.

Continuations have various usages. They may be used as escape procedures to handle exceptions *à la* `setjmp/longjmp` i.e., they are only used while in their dynamic extent. Continuations may also be used for couroutines [19] to interleave multiple subcomputations: these continuations are invoked at most once. Other uses of continuations cover debugging where one wants to be replaced in the exact erroneous context.

Good examples of a continuation multiply invoked are not common in a sequential context. This is no longer true when concurrency is present (as with our `fork` primitive) or, when users ask for bookmarked continuations.

3.1 One-shot use

In [18], Touchette presents the problem of a web transaction with a final commit on a data-base. It is important not to commit twice or the user may pay twice for a single item or two items may be shipped while only one is paid. To forbid re-invocation of past continuations is a first line of defense. This policy is very simply expressed with continuations, we only require the web computation to use the `show-once` function instead of the `show` function:

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

When a page is displayed, its continuation may be resumed only once with success (this code uses the atomic swap effect of the assignment to ensure that only one thread will take the "else" branch of the alternative). Other resumptions will force the current thread to commit suicide. Since it is more user-friendly to tell the user why it is forbidden to re-invoke a past continuation, one may prefer the following alternate definitions:

```
(define (cul-de-sac page)
  (show page)
  (cul-de-sac page) )
(define (show-once page)
  (let ((already? #f))
    (let ((request (show page)))
      (if (set! already? #t)
          (cul-de-sac (message-page "No!"))
          request ) ) ) )
(define (message-page txt)
  (lambda (kUrl)
    (html (head (title txt))
          (body (strong (p txt))) ) ) )
```

Here, we program an endless loop serving a page again and again. A local boolean is freshly regenerated every time `show-once` is invoked and immediately captured by the continuation of `show` where any resumption will find it. Notice that this definition of `show-once` is very safe since it clearly resists to the use of the "Back" button, to window cloning or URLs copy/pasting. To unregister the continuation, that is to suppress its association with its URL, is not as safe since *(i)* the program would not be able to answer gently to such an obsolete URL, *(ii)* other registered continuations may still induce the evaluation of the expression that should not be re-evaluated.

Another more sophisticated manipulation of continuations will be shown in Section 4.2.

3.2 Child-less

While concurrency seems unavoidable, it also introduces

[5]Other snippet of JavaScript may be run in the next answers to kill this window named `"UselessWindow"`. We may also, from the beginning, use hidden frames.

new problems per se. Consider the following program:

```
(fork (suicide)
      (begin (sleep 5) (suicide)) )
```

This program runs for five seconds or so and shows nothing in result. There are three main options for the server *(i)* do nothing, don't even close the connection but wait until the user is exasperated and closes that connection from the browser-side. *(ii)* close the connection so the user's browser may display a "missing data/empty page" warning message. *(iii)* embeds the whole original computation, say π, within the following wrapper to ensure that at least one page will be produced. The wrapper checks to see if a page had been produced (it would have then closed the connection) and outputs a page if that was not already done.

```
(begin π
       (if (open? (current-connection))
           (cul-de-sac
            (message-page "Not much to say!") )
           (suicide) ) )
```

We will see in Section 4 that all these options are useful depending on the situation.

3.3 Orphan

While some computations yield no result, other computations may yield more than one result. This is the case of the following program:

```
(fork (show page1)
      (begin (sleep 10) (show page2)) )
```

In this program, the first `show` expression produces a page and closes the connection. Ten seconds after or so, a sibling thread wants to produce a second page as result but the connection is closed and cannot be used again. The browser does not know that there is another answer while the server has yet another reply to an already replied request. This problem is a kind of dual problem of Section 2 where the server wants now to reply multiply to a single question of the user.

The first idea is to wait for the end of the computations induced by a request that is, the end of the initially created thread as well as the end of all the threads of its offspring. This is a bad solution since it prevents a number of useful tricks: for instance — forking a lurking daemon to release resources after 10 hours of inactivity or — creating an agent waiting for some rendez-vous with another user, or — forking two threads to solve a problem with two different algorithms: a fast heuristical one and a slow but sure one, etc. Moreover, as we said before, it is difficult to detect the end of these computations in presence of first-class continuations.

A second solution is to use the connection as soon as a page gets ready but when `page1` is output, the connection is closed. So we desperately need another open connection towards the browser in order to fill it with `page2`. But if we steal such a connection elsewhere, we just translate the problem to other threads. The solution is clearly to output more than one page in a connection and this is indeed possible.

The solution is to use a snippet of JavaScript to force the browser to open new windows for our supplementary pages. When a page should be replied to a closed connection, the server places it into a queue of waiting pages, see Figure 2. When the server outputs a page on an open unrelated connection, it inserts some JavaScript'ing to instruct the browser to open new windows to host the waiting pages.

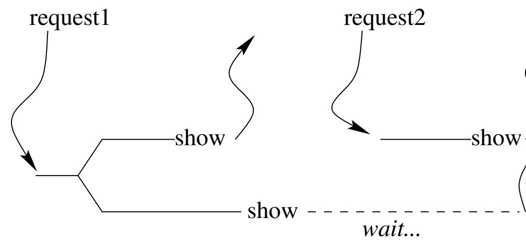


Figure 2: Waiting pages

Here is the snippet[6] which looks a lot like the previous snippet:

```
var pendingUrl;
// Repeat for all pending pages:
pendingUrl = ...;
window.open(pendingUrl,
            "PendingWindow_987715794",
            "");
```

In fact, to avoid delaying pages until a connection is about to be closed, we prefer the following scheme where, as soon as a server accepts a request from a browser for which the server has waiting pages to send, the freshly created thread is immediately reified into a continuation k and the connection is immediately used (see Figure 3) to send the waiting pages accompanied with a redirection resuming k. Of course, when k outputs a page, waiting pages are also checked and sent if any. Waiting pages will therefore wait less.

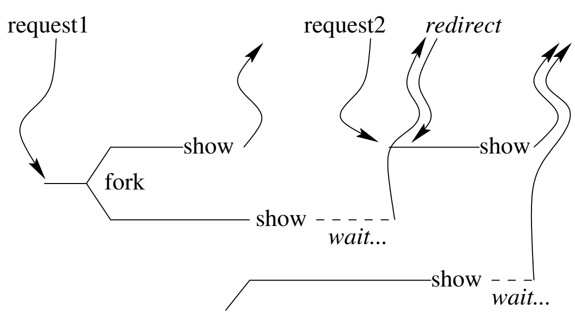


Figure 3: Waiting pages — improved solution

There are some important security points to check for that solution to work since servers may fundamentally only distinguish machines or browsers but not users. Servers should not send old waiting pages to non-vetted users. This may be ensured by the server with authentication, session tracking and so forth.

Our solution has two problems. The first is that without further interaction a user cannot know the other pending results. A trick might bring some illusion of asynchronism again with some JavaScript: we can open a small window with a smart flashy icon that periodically asks the server for waiting pages.

The second problem is more dramatic since it resides in the user's brain. There is no such thing as a user's continuation so when multiple result windows pop up, it is usually difficult for the user to recall what was the original matching request. It is then a problem of pages design to make these contexts apparent (see one possible design based on varying background colors in Section 4.1).

[6] The name of the fresh window is generated to be unique to avoid name conflicts.

3.4 Scope

The above Scheme definitions used the `(current-connection)` form to obtain the connection to use to output a page. This Section proposes a new scope mechanism to implement that function.

Though lexical binding is now the rule, while dynamic binding is still marginally useful [17], [13, p. 167], [9], neither one may be used to define the `(current-connection)` form. When a server accepts a request, be it for an evaluation or the resumption of a continuation, it creates a new thread t and associates the connection to it. The thread t may fork new threads and/or commit suicide. Therefore the associated connection should be also available to all the offspring of t (and to no other threads). This scope is unrelated to lexical scope since the shared mutable global environment is not thread-safe and we must ensure that any piece of code, wherever it is, can `show` a page. This scope is unrelated to dynamic scope since it has nothing to do with the evaluation stack: the thread t may be created with a high stack and shrink it before evaluating `(current-connection)`.

Therefore we are compelled to invent a new scope where an information associated to a thread can be accessed from this thread and all the threads of its offspring but from no other threads. We name that scope "thread+offspring".

Information may be retrieved using the `thread+offspring-get` function that takes a name and a default value and returns the associated value if there is one and the default value otherwise. Following the libertarian spirit of Lisp, if thread+offspring scope is used by the server, it should also be usable by the user. Therefore we offer the `thread+offspring-put` function that binds a name and a value in the thread+offspring scope.

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

We assume the server to place the current connection in the thread+offspring scope of the newly created thread. Therefore the `current-connection` may be defined as:

```
(define (current-connection)
  (thread+offspring-get 'connection #f) )
```

The JavaServer Pages [12] (or JSP for short) proposes a number of specific scopes. Our 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 answer(s) of a request but JavaServer pages limit the use of that information to the first answer while we allow it until threads commit `suicide`. Core Java has a related concept named `java.lang.InheritableThreadLocal`.

Rather informally, our thread scope may be viewed as a dynamic binding that wraps the invocation of the continuation. To evaluate `(thread+offspring-put 'n 'v)` with continuation k in direct style is similar to the following expression where k has been CPS-converted (the `dynamic-let` special form binds n to the value v during the computation of its body; the dynamic value of n may be retrieved with `(dynamic n)`):

```
(dynamic-let ((n 'v))
  (k #unspecified) )
```

In particular, the thread+offspring scope jumps over calls to show i.e., is captured by continuations.

3.5 Life-time

Once an URL appears somewhere in an HTML page and since it may be memorized or retyped any time after, the associated continuation becomes a root for the GC (Garbage Collector) and never disappears. This is clearly unreasonable and servers implement a programmed oblivion.

Most servers offer a "session" object to track the user and record associated information. The session object has a limited life-time say, 2 hours: if a user does not submit request for 2 hours, then the related session object disappears. Registering continuations within the session object ensures them a limited life-time.

This has a side-effect: since session objects are not shared, continuations cannot be shared by this means. Continuations which might be used by more than one user, should be registered elsewhere. For instance, a game where two players exchange their positions may be implemented by a simple exchange of continuations. If continuations are registered in a global table within a server, time-stamps or equivalent might be added to dispose of old continuations.

4. APPLICATIONS

In this Section we describe two applications of our web-continuations. These applications are mostly written in Java, Scheme being the glue.

4.1 PS3I

Our first application is a browser-operated, multi-user, multithread Scheme interpreter named PS3I standing for the Persistent Server-Side Scheme Interpreter. The implemented language is plain Scheme plus `fork` and `suicide` special forms, thread+offspring scope, etc. This interpreter is written as a component in Java.

The PS3I component is configured with respect to the special forms it implements. Then it is possible to create several `PS3I.World`s from this raw component after specifying the Scheme libraries that each `PS3I.World` should load. A `PS3I.World` is essentially characterized by a predefined lexical global environment. Once a `PS3I.World` is created (and this may take time to load all the required libraries), it is possible to create (in a snap) `PS3I.Evaluation`s within it. A `PS3I.Evaluation` is first filled with evaluations (i.e., expressions to evaluate) or resumptions (i.e., continuations to resume) and then run (with one Java thread for each evaluation or resumption). The global environment of a `PS3I.Evaluation` extends the global environment of the `PS3I.World` that gave birth to it. The global environment of a `PS3I.Evaluation` is local to that `PS3I.Evaluation` thus isolating multiple users. This global environment is mutable and may also be extended with new definitions. It is possible to listen to a `PS3I.Evaluation` and detect when all its threads are finished. Several `PS3I.Evaluation`s may concurrently run over a same `PS3I.World`.

A `PS3I.World` is persistent and may be serialized into a file and so are environments and continuations. On the other hand, a `PS3I.Evaluation` corresponds to a collection of running threads and is not serializable.

LAML pages are evaluated within fresh `PS3I.Evaluation`s based on a `PS3I.World` containing the LAML library. Users of PS3I may submit evaluations or resumptions: a request for evaluation creates a new `PS3I.World` offering R4RS libraries, while a request for resumption fetches and re-uses the `PS3I.World` in which the continuation to resume was captured (in order to find the appropriate global environment). When a `PS3I.World` is created, a background color is chosen to ease distinguishing windows belonging to this `PS3I.World` from others. Observe that, contrarily to a classical interpreter with a sequential toplevel loop where every program is evaluated in a shared global lexical environment, we create a new `PS3I.World` for every evaluation (see the second application, the CD-ROM, for a different usage).

There are two fine points dealing with input/output on the default input and output ports. The `display` function, invoked with a single string argument, normally displays this string on the current output port but no such port exists in PS3I. Our implementation just accumulates displayed strings in the current `PS3I.Evaluation`. When a page is about to be flushed, these accumulated strings are then output.

The `read` function shows an inversion of control. Normally the server is seen as an entity that replies to user's requests. When a `(read)` form is evaluated, the server requires an expression from the user. The `read` function is implemented as a special page, see first strip. When continued, this page resumes the continuation of the `(read)` form with the read value.

Continuations cannot be directly implemented in Java since there is no equivalent concept. The PS3I interpreter is written in CPS style as a kind of CEK machine [6] extended with an environment for dynamic bindings and thread+offspring scope. Continuations are explicitly represented by linked list of frames i.e., Java objects. This representation allows numerous sharing since, as noted in [4], captured continuations are often multiply captured.

4.2 CD-ROM

Last year, we built a CD-ROM (mostly in French) that gathers teaching material related to the C programming language (see the associated site at `http://videoc.lip6.fr/`). Among the many goals of the CD-ROM [14] was to help students for their home-work: the CD-ROM provides some "trails" corresponding to lectures notes and laboratory assignments.

Trails are an old idea [2, 20, 16]. They usually correspond to a list of pages to traverse. Moreover they are associated with tools allowing the user to know his position and to move forth and back on the trail. We wanted to have more complex trails with some branching points depending for instance on the performance of the student to a quiz. We also broadly suggested the students to wander through the CD-ROM (150 Mbytes of pages related to C (history, style guides, FAQ, various lectures, etc. plus some indexes, a search engine and a "roulette" button) and be easily reset on their trail.

Our innovation is that trails are Scheme programs evaluated within a `PS3I.World`, the student's position is a continuation and since that continuation is persistent, the student will be reset onto it the next time the CD-ROM is restarted.

Trails are easy to implement within our framework and naïve sequential trails just look like this:

```
(define (trail1)
  (for-each show
            (list url-page-1
                  ...
                  url-page-N ) ) )
```

Observe that while simple this trail allows to traverse a same page more than once without problems. Each time the same page is presented, it has a different continuation. Traditional trails systems do have problems with that fact because of their page-centric point of view.

Our trails offer the following properties:

1. Trails are linear that is, there is exactly one page that follows any other page but for the last page of the trail. Some runtime branching is still possible though but no `fork` may reply twice to the student.
2. At every page, the student may obtain a map describing the visited pages of the trail. This map does not show what remains to be done on the trail since this might not be computable.
3. Already seen pages may be returned to but no computation is involved there. In other words, the page p' that follows a page p is not recomputed, it is memoized as the successor of page p. This is specially useful not to propose again and again the same exercises or jokes!
4. Since a trail is a Scheme program, fresh trails may be easily downloaded from the site of the university to provide new trails over the already existing pages of the CD-ROM.

The web computation that embodies the trail maintains a global history, a chronological list of records that, for each shown page, memorizes its URL, its [after] continuation (i.e., what to do after this page) but also the [before] continuation immediately preceding the call that `show`s that very page. An additional field in the history record mentions, for each successor page, its [from] continuation i.e., the after continuation of its preceding page. Thread+offspring scope is naturally used to determine this information.

With such an history, it is possible to return to a page and redisplay it (with its before continuation), to jump to a successor page without re-computation and to visualize (parts of) the history sorted along various criteria.

The server of the CD-ROM associates a `PS3I.World` for each user. When a request arrives be it for an evaluation or a resumption, the user is determined, the associated `PS3I.World` is resurrected from persistent memory and the request is handled by a Scheme function within that `PS3I.World`. The URL decoding may thus depend on the instantaneous state of the user. This is in contrast with the previous application where the URL decoding was done in an immutable piece of Java code.

Our current rendering of the trail is to provide another button on the pages of the CD-ROM to go to the following page or to be reset on the last page seen of the followed trail. For that latter goal, we just maintain the last resumed continuation in a global variable of the user's `PS3I.World`. Do not confuse the links the page may offer (and, more specifically, the "Previous, Content, Next" buttons of LaTeX2HTML[7] or HEVEA[8] with the "Following" button of the trail. The server of the CD-ROM inserts a standard banner, after the BODY tag of HTML pages, to manage the trail. This is somewhat intrusive but allowed to instrument every HTML page independently of any tool used to build it.

However we feel that this is too intrusive since it definitely spoils some carefully drawn pages. We envision now to deport the banner into an independent trail-manager window to completely avoid to disturb the HTML page. Served pages are simply instrumented to contain some additional JavaScript code to refresh the trail-manager. Our solution thus provides a nice way to organize a visit of pages everywhere on the net without requiring these pages to be aware of this new usage. It will help teachers to re-use already existing material and still provide a commentary about these pages as in Walden's paths [16] but in a more powerful setting.

5. RELATED WORK

The linguistic features of the browser-side are classical, they may be formalized with the web combinators of [3]. The linguistic features of the server-side are also classical. What is original is the parallel that exists between these two sides where `clone` may simulate `fork`.

The concept of web computation where interactions with the user are encapsulated in the `show` primitive is entirely dependent on our use of continuations. Instead of thinking in terms of state and transitions from page to page, we propose an alternate view where a program is suspended and resumed, continuations automatically reifying the state of the computation. This recalls Fuchs' thesis [7] where he advocated the use of continuations for event process. We are in a different setting though because of concurrency.

We introduce the thread+offspring scope to deal with the intermittent input/output streams since neither lexical nor dynamic scope were appropriate. This new scope comes from the various scopes offered by JavaServer Pages [12] except that we adapt it to fit with our (tree-)model of concurrency.

While not simple to master, continuations give a clear and precise view of some disturbing effects produced by browsers. This improves on ad-hoc solution [18] as well as it offers new possibilities such as the one-shot use or the memoizing behavior.

[20] 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 more agile and not page-centric.

Walden's paths [16] 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. Of course the complexity of that programmable view may be lessened with an

[7] From Nikos Drakos, see `http://www-dsed.llnl.gov/files/programs/unix/latex2html/`.

[8] From Luc Maranget, see `http://pauillac.inria.fr/~maranget/hevea/`.

appropriate layer of library functions or macros. This would favor a more declarative definition of trails.

6. CONCLUSIONS

The thesis of this paper is to exhort programmers of web applications to:

1. use a language with concurrency and continuations,
2. adopt a program-centric attitude rather than a page-centric attitude
3. and cope with intermittent input/output streams.

One pleasing outcome of this paper is to give continuations a new, nice and useful rôle. Far from being theoretical ethereal concepts, they stand firm and solve elegantly and efficiently an intricate problem.

Furthermore, this use of continuations provides an interesting approach to expose students to continuations in a clickable way.

PS3I is available under GPL from
`http://www-spi.lip6.fr/ queinnec/VideoC/ps3i.html`

Acknowledgments

Thanks to Emmanuel Chailloux, Luc Moreau, Manuel Serrano and the anonymous referees for all their remarks on this paper. Special thanks to David De Roure who mentioned the word “continuation” when talking of browsers circa 1995.

7. REFERENCES

[1] M. Bieber, F. Vitali, H. Ashman, V. Balasubramanian, and H. Oinas-Kukkonen. Fourth generation hypermedia: Some missing links for the world wide web. *International Journal of Human-Computer Studies*, 47:31–65, 1997.

[2] V. Bush. As we may think. *The Atlantic Monthly*, pages 101–108, July 1945. reprinted in Adele Goldberg (editor), A history of Personal Workstations, ACM Press, New York, 1988, pp 237-247.

[3] L. Cardelli and R. Davies. Service combinators for web computing. *IEEE Transactions on Software Engineering*, 25(3):309–316, May–June 1999.

[4] O. Danvy. Memory allocation and higher-order functions. In *PLDI '87 –ACM SIGPLAN Programming Languages Design and Implementation*, pages 241–252, 1987.

[5] J. D. Davidson and D. Coward. *Java^{TM} Servlet Specification, v2.2*. SUN Microsystems, Dec. 1999.

[6] M. Felleisen and D. P. Friedman. Control operators, the secd-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, Ebberup, Denmark, Aug. 1986.

[7] M. Fuchs. *Dreme: for Life in the Net*. PhD thesis, New York University, Sept. 1995.

[8] R. Kelsey, W. Clinger, and J. Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

[9] L. Moreau. A Syntactic Theory of Dynamic Binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, Dec. 1998.

[10] L. Moreau and C. Queinnec. Design and semantics of quantum: a language to control resource consumption in distributed computing. In *Usenix Conference on Domain Specific Language, DSL'97*, pages 183–197, Santa-Barbara (California, USA), Oct. 1997.

[11] K. Nørmark. Using lisp as a markup language – the laml approach. In *European Lisp User Group Meeting*, Amsterdam, Holland, 1999.

[12] E. Pelegrí-Llopart and L. Cable. *JavaServer Pages^{TM} Specification, version 1.1*. SUN Microsystems, Nov. 1999.

[13] C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996.

[14] C. Queinnec. Enseignement du langage C à l'aide d'un cédérom et d'un site – Architecture logicielle. In *Colloque international – Technologie de l'Information et de la Communication dans les Enseignements d'ingnieurs et dans l'industrie – TICE 2000*, pages 93–102, Troyes (France), Oct. 2000. CNED.

[15] C. Queinnec and D. De Roure. Design of a concurrent and distributed language. In R. H. Halstead Jr and T. Ito, editors, *Parallel Symbolic Computing: Languages, Systems, and Applications, (US/Japan Workshop Proceedings)*, volume Lecture Notes in Computer Science 748, pages 234–259, Boston (Massachussetts USA), Oct. 1993.

[16] F. M. Shipman III, C. C. Marshall, R. Furuta, D. A. Brenner, H.-W. Hsieh, and V. Kumar. Creating educational guided paths over the world-wide web. In *Proceedings of Ed-Telecom '96*, pages 326–331, Boston (Massachussetts USA), 1996. Association for the Advancement of Computers in Education.

[17] G. L. Steele Jr. and G. J. Sussman. The art of the interpreter, or the modularity complex (parts zero, one, and two). MIT AI Memo 453, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.

[18] J.-F. Touchette. Html thin client and transactions. *Dr. Dobb's Journal, Software Tools for the Professional Programmer*, 24(10):80–86, Oct. 1999.

[19] M. Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. The Lisp Conference, 1980.

[20] P. T. Zellweger. Scripted documents: A hypermedia path mechanism. In *Proceedings of Hypertext-89*, pages 1–14, Pittsburgh, PA, 1989.