Inverting back the inversion of control or, Continuations versus page-centric programming *

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

Abstract

Our thesis is that programming web applications with continuations is superior to the current page-centric technology. A continuation is a program-level manageable value representing the rest of the computation of the program. "What to do next" is precisely what has to be explicitly encoded in order to program non trivial web interactions. Continuations allow web applications to be written in direct style that is, as a single program that displays forms and reads form submission since continuations automatically capture everything (control point, lexical bindings, etc.) that is needed to resume the computation. Programming is therefore safer, easier and more re-usable.

1 Introduction

A web application that involves a sequence of interactions with a client (displaying pages and waiting for filled forms) is currently difficult to program because:

1. the client's state in the sequence of interactions may be overwhelmingly complex to encode and maintain,
2. the "Back" and "Clone" capacities of the client's browser may be exercised so that the server may observe multiple (and even concurrent) submissions to already submitted forms.

The latter facts are important to solve since they may deeply impact e-commerce. Their weirdness is only apparent since to go back and fill again a form is a perfectly natural kind of "what-if" programming: try something and if it goes wrong, backtrack and try another thing. To clone a form before filling it is another example of "what-if" programming where the form is preserved before the trial rather than recovered after the trial as with the "Back" button. In the context of e-commerce, a company wants a transaction to exchange exactly n goods for exactly n amounts of money. All other exchanges (two goods for one amount, one good for no amount, no good for one amount, etc.) makes the company or the client unhappy. It is therefore important, when programming a web application that should tolerate these weird facts, *(i)* to detect these supplementary answers, *(ii)* to decide how to handle them.

Our solution is based on the concept of "continuations". Continuations represent "the rest of the computation" or "what to do next?". This is exactly what a tag such as `<FORM ACTION=url>` means that is, where to resume the sequence of interactions. While the mere concept highlights the nature of web interactions, continuations as regular values, allow developpers to program elaborate interactions.

Continuations also offer another definite advantage, they allow developers to program in direct style. In direct style, a web application displays forms to the client and reads values from the client. The interaction is therefore written as a single program (perhaps made of composable subroutines) and not as a series of independent pages sharing some store. Continuations (and lexical closures) capture everything that is needed to perform "what to do next". This capture is entirely automatic: no developper's encoding is required.

The paper first describes the general problem in Section 2 and a running example in Section 3. The concept of continuations is presented in more details in Section 4; problems around the running example are revisited in Section 5, they are solved in Section 6. Some fine points are commented upon in Section 7; related work and conclusion end the paper.

* Revision: 1.13

2 Problems

Browsers provide "Back" and "Clone" facilities. Clients (i.e., users of browsers) may use these facilities to discover the behavior of a site with a "what-if" attitude. From a given form, the client submits some information, examines the result (possibly with some further interaction) and comes back to the original form if the result does not look promising. Instead of going back to the original form, the client may also clone the original form or bookmark it. In all these cases, the client is offered the opportunity to answer again to an already answered form. Even more, cloning allows the client to submit, again as well as concurrently, new answers for already answered forms!

The server has no way to perceive the use of the "Back" and "Clone" facilities; the server has no direct control over the browser. The server may only decode the parameters of the HTTP request in order to detect which form is answered and which action to take given the state of the client. A single interaction that is, one form and at most one answer (such as a poll) is simple to handle since there is no intermediate client's state.

Things become worse *(i)* when a form is made of so many questions that they do not fit in a single page so they have to be spread over several pages, *(ii)* when a form has dependent questions (such as "how many children do you have?" followed by "what are their names?": the generation of the second form depends on the value received from the first form), *(iii)* when the previous navigation of the client influences the following forms ("if the README had been displayed then do not display a link towards it" or "if at least two exercises among some set of exercises were solved in less than five minutes of accumulated time then display a link to the next topic to study").

Historically, interactive programs were monolithic and interact with the client via `print` and `read` functions. The program was blocked on calls to `read`. The uprise of graphical user interface (GUI) provoked a so-called "inversion of control", the program no longer `read`s, it only registers its interest to be resumed with some information (mouse movement, key press, filled form content, etc.). Interactive programs were then built around an event loop that maintains the state of the client and, according to that state, dispatches events to registered subroutines before returning to the event loop with a new, probably modified, state. Multi-threading may ease the implementation of such GUI-based applications by removing the constraint of a single evaluation stack.

Even if the client was in control and no longer the program, the web aggravates that tendency since now the GUI itself is no longer in control but rather delegated to the client's browser which is working offline and synchronizes with the application only when submitting information or displaying resulting pages. When the server displays a form, it has to record "what to do if resumed". This "what to do next" is precisely named a *continuation* in the semantics of computer languages. The Scheme programming language, standardized by IEEE [11791], is the sole standardized language to offer continuations: we will therefore use it.

Using continuations allow the developper to return to direct style and manage continuations where necessary. Instead of seeing a web application as the cooperation of a set of pages, the web application is defined as a single program where continuations take care of sliced execution.

3 Running example

Let us suppose a very simple web application where the client submits a first number, then a second number and gets their sum. This small application will serve us as a running example through the rest of this paper. The involved pages are shown in Figure 1.

Figure 1 illustrates the regular use of that web application. The same information is displayed as a transition matrix in Figure 2. When the server is in the original state (noted $\emptyset$), it displays a first form asking for a number. The number obtained from this form is noted "first number (i)". When i is received, the server switches to state gotN1(i) and displays a second form asking for a second number. The number obtained from this second form is noted "second number (j)". When j is received, a final page is output with the sum of the two numbers. That latter page is not a form so the interaction with the client is finished and cannot be pursued.

The transition matrix of Figure 2 only defines the regular scenario and leaves undefined a number of transitions. How to fill these undefined cells (noted —*x*—) may depend on semantical or pragmatical aspects. Let us review these cells.

cell —a— corresponds to a request yielding a second number while the server is in the original state. The server gets an unsolicited answer for a form it never displayed! This is however possible if a client synthetized such an URL from scratch! To lessen that problem, it is adviseable to use difficultly forgeable URL or cookies.

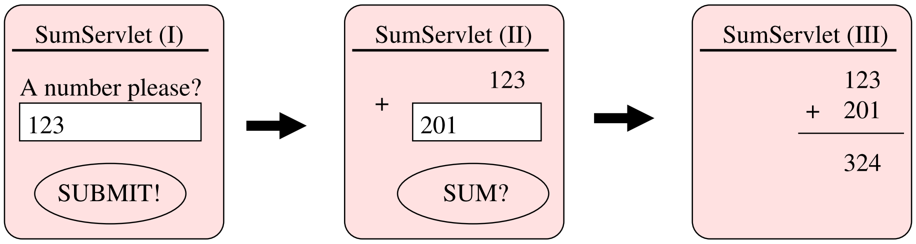

Figure 1: The sum servlet

Server state	first number (i)	second number (j)
$\emptyset$	gotN1(i)	—a—
gotN1(n_1)	—b—	gotN2(n_1, j)
gotN2(n_1,n_2)	—c—	—d—

Figure 2: Regular scenario

A good solution is to define the resulting state to be still the original state but with an additional warning message telling the client that a first number has to be filled in first, see figure 3. Other solutions are also possible.

Server state	first number (i)	second number (j)
$\emptyset$	gotN1(i)	$\emptyset$ /+warning
gotN1(n_1)	—b—	gotN2(n_1, j)
gotN2(n_1,n_2)	—c—	—d—

Figure 3: Unsolicited answer

cell —b— and cell —d— both correspond to a second (or third, or …) submission coming from the previously displayed form. Or the client used the “Back” button to submit again or the client cloned the previous form and submitted more than once from it. A simple solution is to minimize programming and to program only one behavior for any given input. With that goal in mind, the transition matrix may now appear as in Figure 4.

cell —c— corresponds to a second (or third, or …) submission for the first displayed form asking for a first number while a second number had already been submitted. The client may have used twice the “Back” button and re-submitted a new first number. What to do there is unclear. Minimizing the code as before is an obvious solution. However, the submission may be also very naturally understood as an instance of a “what-if” behavior.

Cell —c— may be understood as gotN1(i) (as in the minimizing code solution) corresponding to the cancelation of the second number that is, a backtrack to the first form.

Cell —c— may also be understood as gotN2(i, n_2) that is, a backtrack followed by a replay where the second number is considered to be implicitly submitted again. This behavior is not so bad because a dialog asking for a name then for a huge number of tax information should allow the client to go back and fix a typo in the name without loosing all the tax information already entered.

These variations are not the only possible ones. At least three other variations will be presented in Section 5.

Server state	first number (i)	second number (j)
$\emptyset$	gotN1(i)	gotN2(0, j)
gotN1(n_1)	gotN1(i)	gotN2(n_1, j)
gotN2(n_1,n_2)	gotN1(i)	gotN2(n_1, j)

Figure 4: Minimizing code scenario

4 Continuations

Continuations [SW74, Rey93] were invented for denotational semantics to express the "rest of a computation" in order to give a formal meaning to the `goto` instruction. The concept was so powerful that it had been used to express the semantics of non local jumps, exception handling, coroutines etc. Continuations were introduced in Scheme from the beginning. They were used to let developpers program their own non local jumps, exception handling, coroutines, etc. [Wan80, HFW84].

Consider the following program corresponding to our running example.

IMPLEMENTATION A

```
(define n1 0)
(set! n1 (read-first-number))
(display-result-page n1 (read-second-number n1))
```

This program declares a variable `n1` initialized with 0 then, sequentially, reads a first number and assigns it to `n1`, reads a second number and displays a result page with the sum of these two numbers. The `read-first-number` ships to the client an HTML form asking for the first number. When an answer to that form is obtained (i.e., from the server's point of view: when an HTTP request comes in) the server extracts an integer and assigns it to the `n1` variable. The `read-second-number` function generates an HTML page asking for a second number (making possibly some use of `n1` as shown on Figure 1) and ships it to the client. When an answer to that second form is obtained, the server extracts an integer and yields it as the second argument to the call to the `display-result-page` function.

The continuation of an expression represents what to do with the value of that expression that is, the rest of the computation the expression is involved in. The continuation of the call to the `read-first-number` function may be represented by the following program context where $\boxed{\square}$ represents the exact place where a number is waited for.

```
(set! n1 [□])
(display-result-page n1 (read-second-number n1))
```

Suppose that the number 123 was submitted then the continuation of the call to the `read-second-number` function may be represented by the following context where `n1` is already known to be 123.

```
(display-result-page n1 [□]) |n1=123
```

A context waits for a value and will perform some computation if such a value is injected at the $\boxed{\square}$ place. A continuation may therefore be represented by a unary function. In Scheme, functions are represented by the λ notation. The previous continuation may therefore be represented by:

```
(lambda (n2) (display-result-page n1 n2)) |n1=123
```

That function is technically called a "closure" in the world of functional languages since it contains a reference to the free variable `n1` which was bound to the value 123 when the closure was created.

Our thesis is: *When an HTML page is shipped to the client and that page contains a SUBMIT (or NEXT) button (or link) allowing the client to resume the dialog (i.e., submit information or follow the link), this button (or link) is bound to an URL associated to the continuation of the function that displayed that HTML page.*

For example, the URL bound to the SUM? button (cf. Figure 1) is associated to the continuation of the `read-second-number` function that is:

```
(lambda (n2) (display-result-page n1 n2)) |n1=123
```

Even if Scheme is the sole programming language to offer continuations as first-class values, continuations are everywhere. Any expression of any language has to be evaluated in some environment mapping variables to values and with some continuation telling what to do with the obtained value. Consider the following Java snippet:

```
{ int n1 = 0;
  n1 = read_first_number();
```

```
  display_result_page(n1, read_second_number(n1));
}
```

The continuation of the call to the `read_second_number` method is roughly similar to before and may be represented as:

```
{
  int n1 = 123;
  display_result_page(n1, □);
}
```

Where Java differs from Scheme is that this context cannot be simply turned into a Java value: the developer has to devise a precise encoding. Continuations do exist but are hidden inside the JVM, for instance, the Java scheduler internally uses (kind of) continuations to suspend or resume threads.

In Scheme, continuations may be captured by the `call/cc` primitive function, this name stands for "call with current continuation". To show how it works, let us present how a function such as `read-second-number` is implemented.

```
;;; number → number
(define (read-second-number n1)
  ;; url[ako string] → html[ako string]
  (define (html-generator kUrl)
    (html (head (title "SumServlet (II)"))
          (body (form method: 'post action: kUrl
                      (table (tr (td)     (td n1))
                             (tr (td "+") (td (input type: 'text
                                                     size: 10
                                                     name: "n2" ) )) )
                      (input type: 'submit
                             value: "SUM?" ) )) ) )
  (let ((httpRequest (show html-generator)))
    (string->number (get-request-parameter httpRequest "n2")) ) )
```

The `read-second-number` function defines a generator of an HTML page with the LAML technique [Nør99] that is, with Scheme functions named after HTML tags (one may also use Kiselyov's technique [Kis00]). This generator makes use of the number `n1` and the URL (held in the `kUrl` variable) of the continuation of the caller of the `read-second-number` function. One may also adopt JSP (Java Server Page) or PHP4 pages for that generation provided the generated page contains appropriately `n1` and `kUrl`. This HTML generator (the `html-generator` function) is then given to the `show` library function. The `show` function will return the httpRequest object from which is extracted the *n2* request parameter (as well as converted from a string to a number).

```
;;; (url[ako string] → html[ako string]) → httpRequest
(define (show html-generator)
  (call/cc
    (lambda (k)
      (display (html-generator (k->url k))
               (current-connection) )
      (close-output-port (current-connection))
      (suicide) ) ) )
```

The `show` function belongs to the common library for web applications. It reifies the current continuation into a value named `k`, this continuation is then turned into an URL using the `k->url` function (cf. SubSection 7.1), that URL (say, a string) is given to the HTML generator, the resulting HTML (say a string) is written onto the connection to the client then the current thread (within the server) finishes.

We suppose, for ease of explanation, that the server uses one thread for one incoming HTTP request. When such a request occurs, the server packs the HTTP information into an httpRequest value (say, the union of the `ServletRequest` and `ServletResponse` objects of the servlet world [DC99]) and processes that `httpRequest` value as follows:

```
(define (server httpRequest)
  (let ((url (get-request-url httpRequest)))
    (let ((k (url->k url)))
```

```
(if k
    (k httpRequest)
    (error "lost continuation" url) ) ) ) )
```

The URL used by the request is turned back, using `url->k`, into a continuation. If that continuation is found then the request object (with its query parameters, response stream, etc.) is thrown to it thus making the call to `show` to return with the request object. For C developers, this is quite similar to `setjmp–longjmp` mechanism except that continuations are not restricted in Scheme: once captured, they can be invoked any time and any number of time.

5 Discrepancies

With the implementation A (see above) of our running example, we may observe the correct behavior illustrated in Figure 5. First, the client clones (step A) the first form, submits (step B) 123 as first number, submits (step C) 201 as second number, goes back (step D) to the second number and submits again (step E) another second number: 21. The client then switches (step F) to the cloned first form, submits again (step G) 100 as a new first number then, submits again (step H) 66 as a new second number. Everything looks fine.

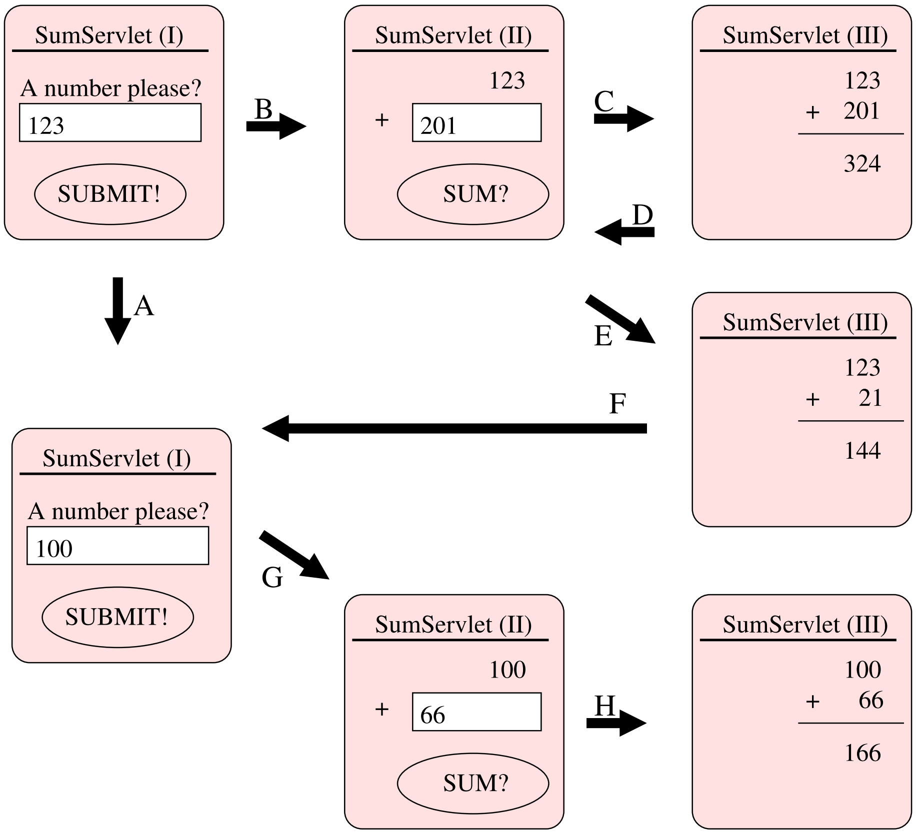

Figure 5: A non problematic scenario

Suppose now that the order of steps is changed as follows (see Figure 6). First, the client clones (step A) the first form, submits (step B) 123 as first number, switches (step C) to the cloned first form, submits again (step D) 100 as a new first number then switches back (step E) to the previous second form, submits (step F) 201 as second number. The displayed addition looks terribly wrong since 123 + 201 is reported as 100 + 201.

In this kind of web application (an educational application for young children explaining the addition), one wants

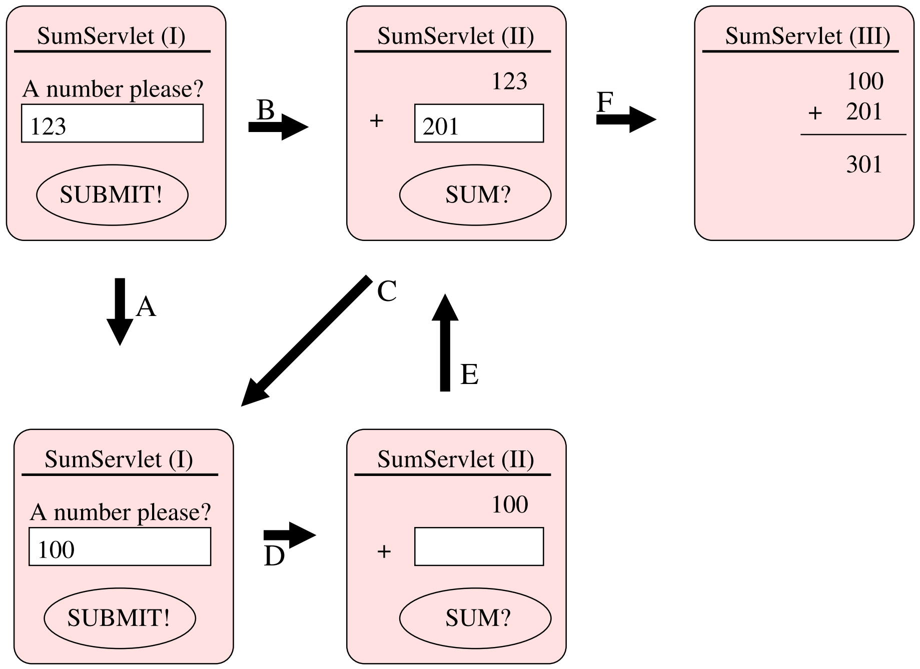

Figure 6: A problematic scenario

probably sums to be always correct. The reason of the discrepancy is the fact that the `n1` variable of implementation A is global and shared among all continuations, therefore, step D alters `n1` from what it was after step B.

There are at least two solutions.

5.1 Enforcing single submission

The first solution is to forbid submitting new answers to already submitted forms thus forcing clients to exactly follow the foreseen scenario. To forbid these new answers just requires to let the calls to `read-first-number` and `read-second-number` to only return a value once. When the client tries to submit another answer, the web application signals it with a warning page.

A simple solution is to replace calls to the `show` function with calls to the `show-once` function.

```
;;; (url[ako string] → html[ako string]) → httpRequest
(define (show-once html-generator)
  (let ((already? #f))
    (let ((httpRequest (show html-generator)))
      (if (set! already? #t)
          (show (message-page "Too late!"))
         httpRequest ) ) ) )

;;; Text[ako string] → (url[ako string] → html[ako string])
(define (message-page txt)
  (lambda (kUrl)
    (html (head (title txt))
          (body (strong (p txt))) ) ) )
```

We suppose the assignment of Scheme to atomically update the involved variable and return its former content. This is a legal behavior in Scheme while not required by the standard. The `show-once` makes use of the atomic swap effect of the assignment. Therefore toggling the `already?` boolean only returns false (i.e., `#f`) once and shows a message page for all other values coming from (show html-generator).

5.2 Making submissions independent

The previous implementation of our web application is very restrictive so another solution is in order. A second solution allows multiple answers but confers them a natural semantics.

Observe the case of the second argument of the call to the `read-second-number` function: every new second number triggers a new call to `read-second-number` independent of all previous calls (this is why step C, D and E) on Figure 5 were correct). We generalize that observation and remove the need for the global variable `n1`.

We simply re-implement our web application as:

IMPLEMENTATION C

```
(let ((n1 (read-first-number)))
  (let ((n2 (read-second-number)))
    (display-result-page n1 n2) ) )
```

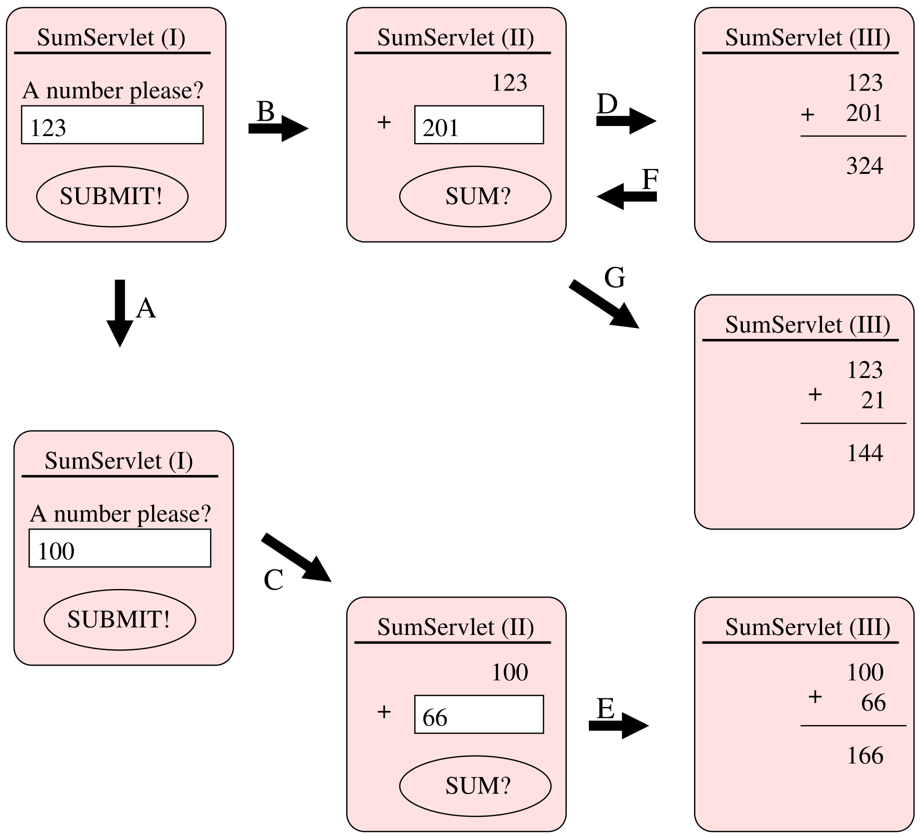

Figure 7: A perfect scenario

This new implementation completely solves the problem. The order of steps does not matter any longer as shown on Figure 7. The function `read-first-number` returns an integer that is stored in a new fresh local variable named `n1`, similarly the function `read-second-number` returns an integer that is stored in a new fresh local variable named `n2`. The continuation of the SUM? button after step B is:

```
(let ((n2 □))
  (display-result-page n1 n2) ) |n1=123
```

while the continuation of the SUM? button after step C is:

```
(let ((n2 [□]))
  (display-result-page n1 n2) ) |n1=100
```

The two pages displayed after steps B and C look similar but, in fact, they differ on the continuation (i.e., the URL) bound to their SUM? button. While based on a similar code, the two continuations possess their own different local `n1` variable. Observe that sharing is not forbidden but mastered: step D and G resume the same continuation and thus share the same local `n1` variable even if creating their own local `n2` variable (this will be more apparent in variation 6.2).

Observe that the concept of a client's state is obsolete, *the continuation is the state* and a client may have multiple waiting continuations at the same time i.e., be in multiple states at the same time: one state for every displayed page.

Observe the elegance of implementation C, the neat change was to use local variables instead of global variables and to let continuations reify what is needed for their resumption both on the control side or the data side.

6 Variations

This section presents some other variations of our web application in order to show how easy they may be programmed.

6.1 Checking form results

Our pages (or html generators as we named them) are viewed as functions taking some arguments and returning some value (we may of course use the "multiple values" features of modern Scheme or gather multiple values in a composite object). To build better re-usable pages, one may require these pages to check the validity of the submitted information.

Here is an example where `read-second-number` is revisited to check that the submitted information is indeed a natural number:

```
;;; number → number
(define (read-second-number n1)
  ;; url[ako string] → html[ako string]
  (define (html-generator kUrl)
    (html (head (title "SumServlet (II)"))
          (body (if warning   ;emit warning if true
                    (font color: "red" warning)
                    "" )
                (form method: 'post action: kUrl
                      (table (tr (td)     (td n1))
                             (tr (td "+") (td (input type: 'text
                                                     size: 10
                                                     name: "n2" ) )) )
                      (input type: 'submit
                             value: "SUM?" ) )) ) )
  (define (loop-until-checked warning)
    (let ((httpRequest (show html-generator)))
      (let ((param (get-request-parameter httpRequest "n2")))
        (if param
            (let ((n2 (string->number param)))
              (if n2
                  n2
                  (loop-until-checked "Not a number!") ) )
            (loop-until-checked "Empty field!") ) ) ) )
  (loop-until-checked #f) )
```

This new definition for `read-second-number` loops until a form submits a non empty parameter that can be successfully parsed as an integer. If these requirements are not satisfied, the same page is displayed with a red warning. The internal `loop-until-checked` function is *tail recursive* that is, preserves its continuation.

6.2 N-ary summing web application

It is straightforward to enhance our web application to the sum of a series of numbers and still allow clients to come back and forth as shown in Figure 8.

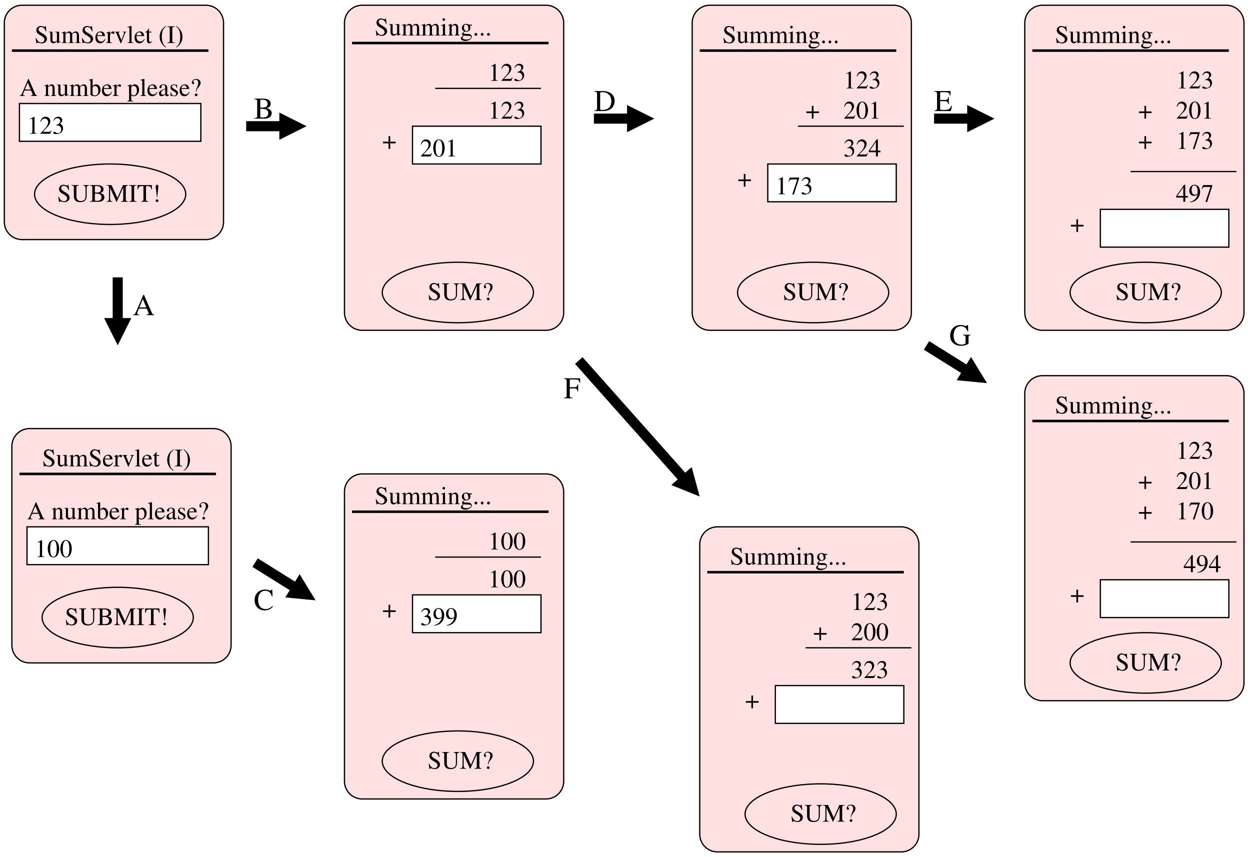

Figure 8: Sum of multiple numbers and backtrack

In Figure 8, the order of steps does not matter much. A new answer to an already answered question spawns a new branch independent of all previous branches. The client may clone or backtrack branches at will and without conflicts. This makes possible to answer a quiz with all its possible answers (if bounded) and see what happens!

Here is the code of this new web application. Instead of asking for two numbers and displaying their sum, a recursive function, named `accumulate`, is provided. The form asking for a new number displays the sum of all previously acquired numbers (gathered in a list).

```
(define (accumulate n1 numbers)
  (let ((n2 (read-again-number n1 numbers)))
    (accumulate n1 (cons n2 numbers)) ) )

(define (read-again-number n1 numbers)
  (define (html-generator kUrl)
    (html (head (title "Summing..."))
          (body (form method: 'post action: kUrl
                      (table (tr (td)     (td n1))
                             (rows numbers)
                             (tr (td) (td "---------"))
```

```
                              (tr (td) (td (apply + n1 numbers)))
                              (tr (td "+") (td (input type: 'text
                                                      size: 10
                                                      name: "n2" ) )) )
                       (input type: 'submit
                              value: "SUM?" ) )) ) )
  (define (rows numbers)
    (if (pair? numbers)
        (string-append (tr (td "+") (td (car numbers)))
                       (rows (cdr numbers)) )
        "" ) )
  (let ((httpRequest (show html-generator)))
    (string->number (get-parameter httpRequest "n2")) ) )

(accumulate (read-first-number) (list))
```

Observe that we use the old `read-first-number` function to ask for the first number. When `read-first-number` function captures its continuation, that continuation now contains the embedding call to `accumulate`. This change of continuation does not impact the definition of the `read-first-number` function. Our pages are re-usable and do not depend on the invocation context.

6.3 Replay

We mentioned replay as a possible behavior for the server when getting another first number after a second number was previously submitted. Such a web application may be sketched as:

IMPLEMENTATION D

```
(define n1 0)
(define n2 #f)        ; initialized to the boolean false
(set! n1 (read-first-number))
(if (not (integer? n2))
    (set! n2 (read-second-number)) )
(display-result-page n1 n2)
```

In this implementation, a new answer to `read-first-number` will re-use the former value of `n2` (if that value is an integer). However, this implementation is only a sketch since it does not ensure that `n2` is only initialized (i.e., set to a value different from the boolean false) once. To ensure that property requires some sort of mutual exclusion which is possible to program with `set!` and `continuation` but is out of the scope of this paper.

6.4 Never regress

Another variation uses a more elaborate use of continuations. In this variation, while the client is allowed to submit to past forms, the submitted information is always used as if coming from the last displayed form. The client is therefore always put back to the last page of the dialog. This new implementation is as follows:

IMPLEMENTATION E

```
(define there #f)
(define (progress f . args)
  (call/cc (lambda (k)
             (set! there k)
             (there (apply f args)) )) )
(define n1 0)
(set! n1 (progress read-first-number))
(set! n2 (progress read-second-number n1))
(progress (lambda () 'nothing))
(display-result-page n1 n2)
```

The trick is the three calls to the `progress` function. This function grabs the current continuation, stores it in a global shared variable (i.e., `there`), the real work is then performed (by applying `f` on its arguments `args`. When this real work returns a value (an httpRequest submitted by the client), this value is thrown back to the `there` continuation. Submissions to past forms will then be handled by the continuation of the last call to `progress`. See Figure 9 for an example.

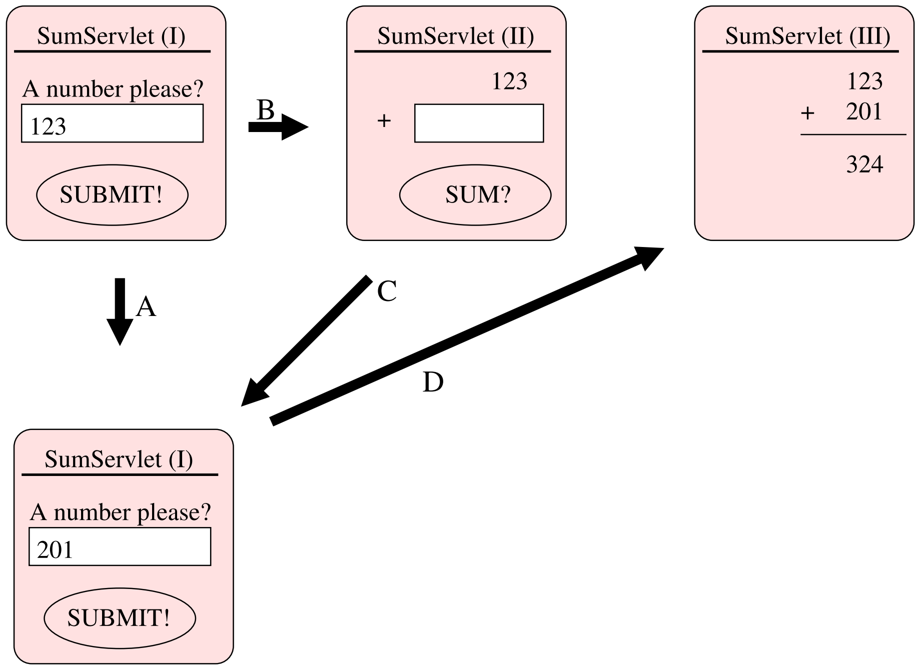

Figure 9: Always back at end

In Figure 9, the client clones (step A) the first form, submits (step B) 123 as first number, switches back (step C) to the cloned form, submits (step D) 201 as new first number. This number is thrown to the continuation of the second form therefore 201 becomes the second number so the final page displaying the sum is shipped.

7 Questions

7.1 How continuations are turned into URLs ?

There are two non exclusive possibilities. The first is to store continuations in the server. For every continuation a difficultly forgeable URL is created, the continuation may then be stored in a hashtable with that URL. The server may store this hashtable in the session object of the client. This confers to these continuations the extent of the session object but it forbids clients to exchange continuations as in a chess game where two opponents may exchange their position. If the server uses a global hashtable then some kind of time-stamping should allow to remove unused continuations after a certain time.

A second possibility is to store the continuation in the client. This may be done in the URL if the continuation is small enough or, alternatively, inside an hidden field of the displayed page. The continuation cannot be stored in a cookie since cookies are shared by all windows of the client's browser.

7.2 How big are continuations ?

That depends on the complexity of the web application (in terms of embedded functions calls). A continuation may roughly be seen as a copy of the evaluation stack (of course there are far better ways to implement them, see [CHO88, HDB90]). Most often, stacks are not very deep. Moreover, stacks may be implemented as linked activation frames where sharing is not only possible but widely observed [Dan87].

We implemented a Scheme interpreter in Java, named PS3I[1], with full continuations for an educational CDROM

[1] `http://www-spi.lip6.fr/~queinnec/VideoC/ps3i.html`

[Que00b]. To reify the control part was not a problem: it amounts for less than 1K byte. The biggest overhead was due to the data part since we had to record the state of the whole global environment and this amounted for 100K. This was because *(i)* the server was running various web applications with different global lexical environments at the same time, *(ii)* global environments are mutable (and may differ among clients of the same web application) so they cannot be shared, *(iii)* the serialization was careless and serializes everything even the immutable predefined shared global Scheme environment that contains a lot of standard but unused functions, *(iv)* finally, we also wanted to check (with additional Scheme programs) students' state to see whether they accomplish their assignment.

For the running example, the continuation should contain the program of the web application as well as the global variables it requires and this should only amount to a few Kbytes or a few bytes if the web application is considered constant (see [Hug00]).

8 Related work

The first time I heard of "continuations" in the context of browsers was from David De Roure circa 1995: he qualified the "Back" button as the invocation of some continuation. The motivation for that study was the design of an educational CDROM (around the C programming language) where I wanted to offer many links to many C-related resources (guides of style, history, programs, etc.) to encourage wandering through the culture around C but I still wanted pages to display a button that, when hit, brings students back onto their assigned trail (a set of pages to read and practice). The implementation of that button is based on continuations [Que00b] along the lines of implementation E.

Dreme, the PhD thesis of Matthew Fuchs [Fuc95] was, to the best of my knowledge, the first mention that event loops may be encoded with continuations. We further this idea for web interactions.

MAWL [LR95] standing for – The Mother of All Web Languages – is an application language for programming interactive services in the context of the World Wide Web. We share a lot with MAWL: pages are viewed as functions and continuation urls are synthetized. However we differ from MAWL on a number of points: continuations are handled explicitly (and this allows to build more sophisticated interactions), we use an existing language, Scheme, rather than a specialized language (and we loose static analyses such as type checking). However our improvements may be incorporated to other languages provided they offer continuations.

JF Touchette [Tou99] addresses the problem of the non-repeatability of transactions in presence of the "Back" and "Clone" buttons. Provided you master continuations, we believe our solution to be more robust, more systematic and more customizable.

Monads are strongly related to continuations. Recently [Hug00], monads were generalized into arrows. One application, envisioned by John Hughes, 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 solves the problem of multiple submissions from the same form since submissions bring the state along which they should be considered. 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 our scheme is independent on which side records continuations, we tend to record them in the server as parts of the state of a whole interpreter which we may examine to check whether students accomplish their assignment.

9 Conclusions

Our contribution is two-fold:

1. A Web application should be written as a single program in direct style and not as a series of separate pages where the state and the continuation have to be explicitly encoded. We bring, after MAWL, another evidence of that claim.

2. Continuations are a useful concept to build sophisticated interactions and web languages should offer them.

We continue to explore the use of continuations for our educational web applications [Que00a]. Since continuations are only useful in some specific and narrow places, we are currently implementing a library of functions to mostly hide their use. This library will allow developers to specify, in a declarative way, the kind of scenarii (i.e., replay, non-regress, etc.) they have in mind.

References

[11791] IEEE Std 1178-1990. *Ieee Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[CHO88] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for continuations. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 124–131, August 1988.

[Dan87] Olivier Danvy. Memory allocation and higher-order functions. In *PLDI '87 –ACM SIGPLAN Programming Languages Design and Implementation*, pages 241–252, 1987.

[DC99] James Duncan Davidson and Danny Coward. *Java™ Servlet Specification, v2.2*. SUN Microsystems, December 1999.

[Fuc95] Matthew Fuchs. *Dreme: for Life in the Net*. PhD thesis, New York University, September 1995.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, White Plains, New York, June 1990.

[HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 293–298, Austin, TX., 1984.

[Hug00] John Hughes. Generalising monads to arrows. *to be published in "Science of Computer Programming"*, 2000?

[Kis00] Oleg Kiselyov. Implementing metcast in scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, volume Rice COMP TR00-368, pages 23–25, Montréal (Canada), September 2000.

[LR95] D. Ladd and J. Ramming. Programming the web: An application-oriented language for hypermedia services. In *4th International World Wide Web Conference — WWW4*, pages 567–586, Boston (Massachussetts USA), December 1995. World Wide Web Consortium.

[Nør99] Kurt Nørmark. Using lisp as a markup language – the laml approach. In *European Lisp User Group Meeting*, Amsterdam, Holland, 1999.

[Que00a] Christian 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'ingénieurs et dans l'industrie – TICE 2000*, pages 93–102, Troyes (France), October 2000. CNED.

[Que00b] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ICFP '2000 – International Conference on Functional Programming*, pages 23–33, Montreal (Canada), September 2000.

[Rey93] J C Reynolds. The discoveries of continuations. *International journal on Lisp and Symbolic Computation*, 6(3/4):233–247, 1993.

[SW74] C Strachey and C P Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monography PRG-11, Oxford University, Computing Laboratory, Oxford University, England, 1974. Reproduced in [SW00].

[SW00] C Strachey and C P Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, May 2000.

[Tou99] Jean-François Touchette. Html thin client and transactions. *Dr. Dobb's Journal, Software Tools for the Professional Programmer*, 24(10):80–86, October 1999.

[Wan80] Mitchell Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28. The Lisp Conference, 1980.