Enabling Complex User Interfaces In Web Applications With send/suspend/dispatch

Peter Walton Hopkins
Computer Science Department
Brown University
Box 1910, Providence, RI 02912
phopkins@cs.brown.edu

ABSTRACT

Web application programmers who use PLT Scheme's web server enjoy enormous benefits from the **send/suspend** primitive, which allows them to code in a direct style instead of in continuation-passing style as required by traditional CGI programming. Nevertheless, **send/suspend** has limitations that hinder complex web applications with the rich interfaces expected by users. This Scheme Pearl introduces a technique for "embedding" Scheme code in URLs and shows how this facilitates developing complex web-based user interfaces.

1. USER INTERFACES IN HTML

As web applications become more popular and powerful, the demands on their interfaces increase. Users expect complex interface elements that emulate those found in desktop applications: tabs for switching among screens of data, tables that can sort with clicks on their column headers, confirmation "dialogs," etc.

The screenshot in Figure 1 shows a web application with two such UI elements: a row of tabs across the top and a table with clickable column headers. The page is from CONTINUE [4], a Scheme web application for accepting paper submissions and managing the conference review process. I re-wrote CONTINUE using the technique in this paper, so I will use it as a recurring example of a web application with a rich user interface.

HTML provides several user interface elements (radio buttons, check boxes, text fields, etc.) and web browsers give them an OS-appropriate appearance and behavior. Interface elements not provided by HTML—tabs, for example—must get their appearance from HTML mark-up and their behavior from the web application code. The web application programmer must devote a non-trivial amount of code to emulating complex elements by reducing them to HTML's universal interface element: the hyperlink.

In the screen shot, each tab is a link, each column header is a link, and each paper title is a link. When the user clicks on any of these the browser will send a request to the servlet and the servlet must respond with a new web page that has the appropriate change in interface. This means that the links must be constructed—and the servlet must be coded to recognize the construction—in a way that can convey their meaning. Is it a click on a tab? Or should the columns be re-sorted? Or should an entirely different page (a CONTINUE example is a page showing reviews for a paper) be returned?

The page from CONTINUE shows three very different classes of links: tabs, column headers, and paper titles. This diversity of hyperlinks makes implementing this page with the PLT web server's core primitive, **send/suspend**, difficult. This paper describes an extension to **send/suspend**, called **send/suspend/dispatch**, that vastly simplifies the code necessary for complex pages.

This paper was partially supported by NSF Grants ESI-0010064, ITR-0218973, SEL-0305949 and SEL-0305950, and by Brown University's Karen T. Romer UTRA program.

2. THE PLT WEB SERVER

Because of the nature of CGI—the web application process halts after returning a page to the browser—"traditional" web programming must be written in a continuation-passing style [5, 2]: a web page sent to the browser must contain enough data (commonly in hidden form fields) so that the server can pick up where it left off when it had to terminate.

The PLT web server [3] uses Scheme's first-class continuations to avert a CPS transformation and the problems (unclear program flow, serialization of data into strings, exposure of some application internals) associated with it.

send/suspend is the PLT Scheme web server's primitive for capturing a servlet's continuation. It consumes a page-generating function of one argument: a URL that will resume the continuation, which by convention is named *k-url*. The result of evaluating the page-generating function with a *k-url* is sent to the user's web browser. When a link to *k-url* is clicked the browser makes a request to the servlet and **send/suspend** resumes the continuation by passing it the request.

send/suspend will only capture one continuation per page: the "actual" continuation waiting for the browser's request. But, most web application pages have multiple "logical" continuations pending. The CONTINUE page in Figure 1 has three: one waiting for a tab to switch to, one waiting to sort the list, and one waiting for a paper to show in detail. With **send/suspend**, the programmer must shoehorn a page's logical continuations into the one actual continuation that will be resumed. The code to do this dispatching is both fragile and generalizes poorly.

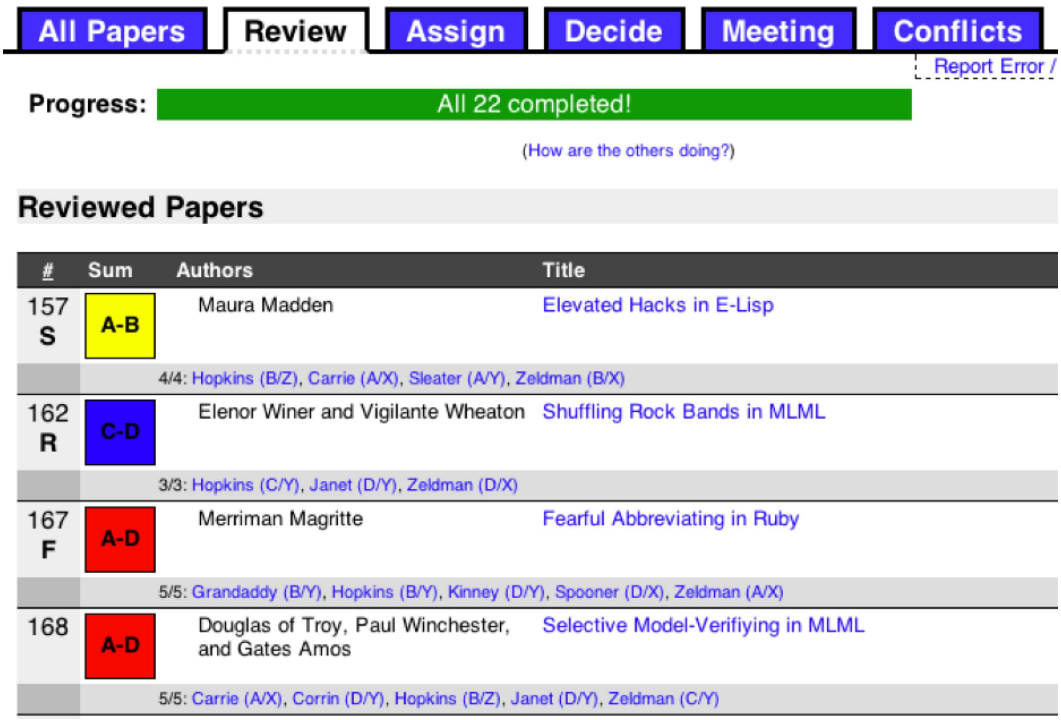

Figure 1: Screenshot from the CONTINUE server

3. DISPATCHING WITH SEND/SUSPEND

In general, **send/suspend** requires the programmer to encode enough data into a page's hyperlinks that the actual continuation can dispatch to the correct logical continuation. HTTP has at least two methods of adding data to a URL, the most flexible of which is to append a query string of the form `?`key_1`=`$value_1$`&`key_2`=...`. The code to do this for the page titles in the CONTINUE example might look like:

```
1 = (send/suspend
       (lambda (k-url)
         `(html ...
           (a ([href ,(format "~a?paper=157" k-url)])
              "Elevated Hacks in E-Lisp")
           (a ([href ,(format "~a?paper=162" k-url)])
              "Shuffling Rock Bands in MLML")
           (a ([href ,(format "~a?paper=167" k-url)])
              "Fearful Abbreviating in Ruby"))))
```

Or, more realistically, with a map over a list:

```
2 = (send/suspend
       (lambda (k-url)
         `(html ...
           ,@(map
              (lambda (paper-num)
                `(a ([href ,(format "~a?paper=~a"
                                    k-url paper-num)])
                    ,(paper-title paper-num)))
              (get-all-papers)))))
```

This encoding solution was proven workable by the first few versions of CONTINUE. All it needs is code to interpret the request from the browser, pick out which paper the user clicked on, and dispatch accordingly:

```
(define (show-list/review)
  (let* ([request 2]
         [bindings (request-bindings request)]
         [paper-num
           (string→number
             (car (extract-bindings 'paper bindings)))])
    (show-paper paper-num)))
```

The servlet environment handily converts the URL's query string into a list of (*key* . *value*) pairs, and it also provides the *extract-binding* function to return all values matching a given key. A call to the *show-paper* function sends the requested paper's page to the user's browser.

send/suspend is perfectly adequate in this situation because there is only one logical continuation to resume: the one waiting for a paper to display. Adding just one additional logical continuation—for example, one that is waiting to switch to a particular tab—will make the above dispatching code much more complicated.

First, here is the HTML-generating code for the page with tabs. URLs are now postfixed by either a `?paper=`n or a `?tab=`n. The class attribute is used to signal that the Review tab is current and should be displayed differently.

```
3 = (send/suspend
       (lambda (k-url)
         `(html ...
           (ul ([id "tabs"])
               (li (a ([href ,(format "~a?tab=1" k-url)])
                      "All Papers"))
               (li (a ([href ,(format "~a?tab=2" k-url)]
                       [class "selected"])
                      "Review"))
               (li (a ([href ,(format "~a?tab=3" k-url)])
                      "Assign"))
               (li (a ([href ,(format "~a?tab=4" k-url)])
                      "Decide")))
           ...
           ,@(map
                (lambda (paper-num)
                  `(a ([href ,(format "~a?paper=~a"
                                      k-url paper-num)])
                      ,(paper-title paper-num)))
                (get-all-papers)))))
```

The dispatching code must now check which logical continuation is being resumed: tab or paper.

```
(define (show-list/review)
  (let* ([request 3]
         [bindings (request-bindings request)])

    (cond [(pair? (extract-bindings 'paper bindings))
           (let ([paper-num
                  (string→number
                    (car (extract-bindings 'paper bindings)))])
             (show-paper paper-num))]

          [(pair? (extract-bindings 'tab bindings))
           (let ([tab-num
                  (string→number
                    (car (extract-bindings 'tab bindings)))])
             (case tab-num
               [(1) (show-list/all)]
               [(2) (show-list/review)]
               [(3) (show-list/assign)]
               [(4) (show-list/decide)]))])))
```

This dispatching code is more involved than the code for the previous case, though extending it for additional logical continuations is straightforward: the programmer adds more HTML to encode more data in a URL, then he adds another clause to the **cond** to handle the new logical continuation. But, though workable, there are two major problems with this pattern of web programming.

The first is that the code to generate the HTML and encode data into the URLs is separate from the code to decode the URLs and perform the dispatching. These pieces of code are tightly dependent, so changes to one must be matched by changes to the other. Their separation imposes a higher maintenance burden to ensure that they are kept in synch.

The second problem is that the programmer cannot easily generalize complex interface elements into their own functions. Though not very not evident from a single example, this is a major limitation for medium-to-large–scale web applications because it breaks down abstractions and forces code duplication.

For example, CONTINUE uses tabs on nearly every page, so a function to create them and handle Their behavior would be useful from a development and a maintenance standpoint. But, **send/suspend** makes such a function impractical. The closest solution is one function that generalizes the HTML and another the behavior, with the understanding that these two must be kept closely in synch:

```
(define generate-tabs (k-url tab-list selected)
  `(ul ([id "tabs"])
    ,@(map
        (lambda (tab-pair)
          `(li (a ([href ,(format "~a?tab=~a"
                                  k-url (car tab-pair))]
                   [class ,(if (equal? (car tab-pair) selected)
                               "selected"
                               "")])
                  ,(car tab-pair))))
        tab-list)))

(define dispatch-tabs (request tab-list)
  (let* ([bindings (request-bindings request)]
         [tab-bindings (extract-bindings 'tab bindings)])
    (and (pair? tab-bindings)
         ((cdr (assoc (car tab-bindings) tab-list))))))

(define (show-list/review)
  (let* ([tab-list `(["All" . ,show-list/all]
                     ["Review" . ,show-list/review]
                     ["Assign" . ,show-list/assign]
                     ["Decide" . ,show-list/decide])]
         [request 4]
         [bindings (request-bindings request)])

    (cond [(pair? (extract-bindings 'paper bindings))
           ...]
          [(dispatch-tabs request tab-list)])))

4 = (send/suspend
       (lambda (k-url)
         `(html ...
           ,(generate-tabs k-url tab-list "Review")
           ...
           ,@(map
                (lambda (paper-num)
                  `(a ([href ,(format "~a?paper=~a"
                                      k-url paper-num)])
                      ,(paper-title paper-num)))
                (get-all-papers)))))
```

Even with this generalized version, the page function is still responsible for dispatching each logical continuation. An interface element cannot be added to a page without both a clause in the dispatch code to handle it and all the necessary dispatching data (in this case, *tab-list*) in scope. With **send/suspend** there is no clear way to have a single, opaque function that adds a set of tabs to a page.

4. SOLUTION: SEND/SUSPEND/DISPATCH

send/suspend/dispatch solves the above two problems by allowing the programmer to specify a continuation, in the form of a closure, for every URL on a web page. This generalizes the encoding,

decoding, and dispatch necessary with **send/suspend**'s single continuation model. By "embedding" closures into the page the programmer can easily write complex UI elements because presentation and behavior are local in the file and generalization of elements into functions is possible.

send/suspend/dispatch presents an interface very similar to **send/suspend**'s: pages are still generated by a function of one argument. But instead of the static *k-url*, page-generating functions receive a function—conventionally called *embed/url*—that consumes a function of one argument and returns a unique URL. The page-generating function uses *embed/url* to embed continuations into URLs. When a URL is requested by the browser the continuation embedded in that URL will be resumed, receiving the browser's request as its argument.

The following is the first example from above, converted to use **send/suspend/dispatch**. A separate dispatching step is no longer necessary; **send/suspend/dispatch** manages resuming the continuation when a URL is accessed.

```
(define (show-list/review)
  (send/suspend/dispatch
    (lambda (embed/url)
      `(html ...
        ,@(map
           (lambda (paper-num)
             `(a ([href ,(embed/url
                          (lambda _
                            (show-paper paper-num)))])
                 ,(paper-title paper-num)))
           (get-all-papers))))))
```

Tabs are added easily to the above code, again with no explicit dispatching:

```
(define (show-list/review)
  (send/suspend/dispatch
    (lambda (embed/url)
      `(html ...
        (ul ([id "tabs"])
          (li (a ([href ,(embed/url
                          (lambda _ (show-list/all)))])
                 "All Papers"))
          (li (a ([href ,(embed/url
                          (lambda _ (show-list/review)))]
                  [class "selected"])
                 "Review"))
          ...)
        ...
        ,@(map
           (lambda (paper-num)
             `(a ([href ,(embed/url
                          (lambda _
                            (show-paper paper-num)))])
                 ,(paper-title paper-num)))
           (get-all-papers))))))
```

The above code examples demonstrate how presentation code and behavior code are local to each other in the file with **send/suspend/dispatch**. The flow of control in these examples is clear because of that locality.

Now we can construct a more useful generalization of tabs. Unlike the previous pair of functions, this function is self-contained: because all dispatching is handled by **send/suspend/dispatch**, the result of *generate-tabs* can simply be dropped into a page and its behavior will be handled properly.

```
(define generate-tabs (embed/url tab-list selected)
  `(ul ([id "tabs"])
    ,@(map
       (lambda (tab-pair)
         `(li (a ([href ,(embed/url
                          (lambda _ ((cdr tab-pair))))]
                  [class ,(if (equal? (car tab-pair) selected)
                              "selected"
                              "")])
                 ,(car tab-pair))))
       tab-list)))
```

The ability to write self-contained functions like *generate-tabs* is essential for adding complex UI elements to web applications. For example, CONTINUE has a general function, *make-paper-list*, to create lists of papers like the one in Figure 1. It can be added to the previous example:

```
(define (show-list/review view-info)
  (send/suspend/dispatch
    (lambda (embed/url)
      `(html ...
        ,(generate-tabs embed/url `(...) "Review")
        ,(make-paper-list
           (get-all-papers) embed/url show-paper
           show-list/review view-info)))))
```

make-paper-list takes a list of papers to show, the *embed/url* function, a function to invoke when a paper is clicked on, a callback to re-display the current page, and data defining how to show the list. To *show-list/review*, *view-info* is an opaque vehicle for passing data back into *make-paper-list* when the callback is used.

Clickable column headers are implemented entirely within *make-paper-list*: they call the callback (*show-list/review* in this case) with *view-info* changed to include the new sort. This will re-display the same page the user was looking at, but the sorting of the papers will be different.

make-paper-list can be extended with logical continuations for additional behaviors (filtering by rating, for example) without any changes to *show-list/review*, *show-list/assign*, or any other function that calls *make-paper-list*. This is possible because the *show-list/*∗ functions do not have to handle any dispatching themselves.

4.1 Is s/s/d A Step Backwards?

One of the most useful features of **send/suspend** is that it prevents a global CPS transformation of web application code. At first glance, **send/suspend/dispatch** looks regressive because it forces the programmer to be explicit with his continuations.

Though code using **send/suspend/dispatch** resembles CPS code, this is acceptable for several reasons. First, the CPS transformation is local, not global. Code is still written in a mostly direct style, and **send/suspend/dispatch**'s embedded continuations are arguably clearer than the separate dispatch **cond** from **send/suspend**

code. Second, in practice the embedded continuations tend to simply call named functions like in the tab example above. This also keeps the flow of control clear. Finally, a multiple-continuation model fits a web page more accurately than a single-continuation model. As the CONTINUE example shows, web pages often have several logical continuations active at one time, and **send/suspend/dispatch** correctly captures that pattern.

4.2 send/suspend/dispatch With Forms

In all previous examples the embedded continuations used the _ argument convention to ignore the value they were resumed with, which is the request data sent from the browser. When the continuation is embedded in a hyperlink the request data is rarely relevant, but it is necessary when the continuation is embedded into the action URL for a form:

```
`(form ([action ,(embed/url
                   (lambda (request)
                     (handle-form-bindings
                       (request-bindings request))))])
  ...
  (input ([type "submit"] [name "button"]
          [value "Save"]))
  (input ([type "submit"] [name "button"]
          [value "Save and Continue"])))
```

The bindings for a request contain the data from the form. A continuation embedded in a form will access these and process them as necessary.

An intrinsic shortcoming of HTML forms—not addressed by either **send/suspend** or **send/suspend/dispatch**—is that each form has a single action URL that form data is always sent to. This makes it impossible to distinguish between different submit buttons using URLs.

Some forms in the CONTINUE server faced this problem. For example, when assigning PC members to review a paper the PC chair has one button to save his decisions and remain looking at the same paper, and another button to save his decisions and automatically show another paper. These two buttons are in the same `form` tag because they need to share the same form elements (checkboxes, etc.). Because they share a form, they share an action URL and, therefore, a single re-entry point in the servlet. The code embedded in the form's action URL must handle the dispatch on which button was clicked:

```
(define (handle-form-bindings bindings)
  (let ([button (extract-binding/single 'button bindings)])
    (save-assignment-data request)
    (cond [(equal? button "Save")
           ;; show same paper
           ]
          [(equal? button "Save and Continue")
           ;; show new paper
           ])))
```

The *handle-form-bindings* function will receive all the data from the form. But it must fall back on **send/suspend**-like dispatching to determine if the user clicked "Save" or "Save and Continue."

If the programmer could specify unique URLs for each button he could use **send/suspend/dispatch** to embed separate functions for each button. With the current state of HTML the only solution to the two-button problem is to use a single URL and examine the request bindings to determine which button was clicked.

5. IMPLEMENTATION

send/suspend/dispatch is defined in terms of **send/suspend**, augmenting it by transparently handling the encoding of *k-url* and the subsequent dispatching. The code for **send/suspend/dispatch** is in Figure 2. Figure 3 contains the necessary helper functions.

First, **send/suspend/dispatch** creates a hash table (*embed-hash*) to store embedded functions. The keys for this hash table are random numbers and are generated by the *unique-hash-key* funcion.

1: **send/suspend/dispatch** calls **send/suspend** to send the page to the browser and get the response (what the user clicked on). *page-func* is the user's page-generating function, which takes *embed/url* as its argument.

When called with no arguments, *embed/url* just returns *k-url*. When called with a function to embed as its argument it uses 2 to generate a unique, random key and store the function (*embed/func*) in the hash table with that key. *embed/url* then calls *url-append/path* on *k-url* and the key to create a URL that will resume **send/suspend**'s actual continuation but carry with it an identifier for the logical continuation to resume.

A **send/suspend** URL (disregarding the `http://` and server) looks like this:

```
/servlets/cont.ss;id313*k2-1167813005
```

The text following the `;` identifies the continuation that **send/suspend** will resume. A **send/suspend/dispatch** URL includes the hash key in the path portion of the URL:

```
/servlets/cont.ss/34412;id313*k2-1167813005
```

Once the browser responds with a request, **send/suspend/dispatch** extracts the key from the URL by calling *post-servlet-path* to get the part of the path following the servlet's extension. The first piece of this path is the key.

3: Finally, **send/suspend/dispatch** looks up the key in the hash table (returning a simple error page if it is not found) to get the continuation embedded in the link the user clicked. It calls this function with the browser's request as an argument.

6. CONCLUSION

send/suspend/dispatch, an extension to the PLT web server's **send/suspend**, vastly simplifies servlet coding by enabling a very natural abstraction: that each URL on a web page is tied to a separate, pending continuation. **send/suspend**'s one-continuation model led to inappropriately divided code, prevented generalizations, and forced the programmer to handle low-level issues of URLs and query strings.

Web pages that had several logical continuations, such as those emulating complex user interface elements, become natural and straightforward to implement with **send/suspend/dispatch**.

```
(define/contract send/suspend/dispatch
  (page-func/ssd-contract . → . any)
  (lambda (page-func)
    (let* ([embed-hash (make-hash-table)]
           [request 1 ]
           [path
            (post-servlet-path
             (url→string (request-uri request)))])
      (if (null? path)
          request
          3 ))))
1 = (send/suspend
      (lambda (k-url)
        (page-func
          (case-lambda
            [() k-url]
            [(embed-func) 2 ]))))
2 = (let ([key (unique-hash-key embed-hash)])
      (hash-table-put! embed-hash key embed-func)
      (url-append/path k-url key))
3 = ((hash-table-get
      embed-hash
      (string→number (car path))
      (lambda ()
        (lambda _
          (send/back
            '("text/plain"
              "ERROR: Key was not found in "
              "send/suspend/dispatch hash table")))))
     request)
```

Figure 2: send/suspend/dispatch Implementation

7. ACKNOWLEDGMENTS

Thanks to Shriram Krishnamurthi for the original CONTINUE code and for invaluable help advising me as I rewrote it and put together this paper.

8. REFERENCES

[1] R. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[2] P. T. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the Web. In *IEEE International Symposium on Automated Software Engineering*, pages 211–222, Nov. 2001.

[3] P. T. Graunke, S. Krishnamurthi, S. van der Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. In *European Symposium on Programming*, pages 122–136, Apr. 2001.

[4] S. Krishnamurthi. The CONTINUE server. In *Symposium on the Practical Aspects of Declarative Languages*, 2003. **Invited Paper**.

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

```
;; hashtable → number
;; given a hash table, finds a number not already used as a key
(define unique-hash-key
  (lambda (ht)
    (let ([key (random 200000)])
      (let/ec exit
        (hash-table-get ht key (lambda () (exit key)))
        (unique-hash-key ht)))))

;; string → listof string
;; takes a string that is the path portion of a URL
;; finds the path that follows the .ss extension, splits it
;; at / characters, and returns the list
(define post-servlet-path
  (lambda (s-url)
    (let ([result (regexp-match "\\.ss(/[^;#\\?]*)" s-url)])
      (if result
          (filter
            (lambda (s) (> (string-length s) 0))
            (regexp-split "/" (cadr result)))
          null))))

;; string any → string
;; adds a datum to the end of the path of a URL represented
;; as a string. The added datum comes before any query or
;; parameter parts of the URL.
(define url-append/path
  (lambda (s-url rel-path)
    (let ([url (string→url s-url)])
      (url→string
        (make-url
          (url-scheme url)
          (url-host url)
          (url-port url)
          (format "~a/~a" (url-path url) rel-path)
          (url-params url)
          (url-query url)
          (url-fragment url))))))

;; contracts for embed/url and page-generating functions,
;; for use with PLT Scheme's contracts [1]
(define embed/url-contract
  ((→ string?) . case→ .
   ((request? . → . any) . → . string?)
   (string? (request? . → . any) . → . string?)))
(define page-func/ssd-contract
  (embed/url-contract . → . any))
```

Figure 3: Helper functions for send/suspend/dispatch