Finite State Machines for the Web

A348/A548 Final Exam Review
December 8, 2003

Abstract

HTTP is a purely functional protocol[1]: you ask a question, you get an answer. But the client browser is a perfect repository for state information: one can use cookies, hidden fields, URL path information, or the memory of the browser process. Sensitive information, however, must be kept on the server, in session variables. Traditional programming implements state machines as a `while/switch` combination. In a web context each HTTP transaction becomes one pass through the loop (which literally disappears from the code) and the body of the loop must be very specific about (a) the information that participates in the change of state and (b) its provenance (client side vs. server side). The pattern is presented below. It is argued[2] that this pattern greatly increases the chances of a beginner student in succeeding as an effective web programmer.

A Basic Scheme

The example below shows how tricky it is to keep state by using only the client side of the equation

```
(define calc
  (lambda (arg acc cont)
    (let ((acc (+ acc arg)))
      (display (list "The accumulator is now " acc))
      (newline)
      (lambda (arg)
        (cont arg acc cont)))))

> (define browser (calc 10 0 calc))
(The accumulator is now  10)
> (define browser (browser 20))
(The accumulator is now  30)
> (define browser (browser -23))
(The accumulator is now  7)
> (define browser (browser 20))
(The accumulator is now  27)
```

[1] Which is effectively responsible for the event-driven nature of the web.
[2] You are my argument so, if you really want, you can prove me wrong.

Essentially `calc` creates a procedure of one argument that reproduces itself with a change of state, for the calculation it has performed. (This is a simple calculator with one accumulator). `browser` could have also been called `new-calc`, or `new-browser`. Here's a ready to be used calculator that has started with an accumulator of 0, after we added 10 to it, subtracted 5, and added 8:

```
(((calc 10 0 calc) -5) 8)
```

Here it is in action:

```
> (((calc 10 0 calc) -5) 8)
(The accumulator is now  10)
(The accumulator is now  5)
(The accumulator is now  13)
 #<procedure>
>
```

The purpose of this paper is to demystify code such as the above and render it useful for the masses in a web programming (HTTP) context. That is, the usefulness and beauty of the code above is unquestionable[3]. But the appreciation lies in the eyes of the beholder, and the beholder (sometimes for a very good reason) may not be in a good mood to appreciate the succint force of such an abstract pattern.

The HTTP Pattern

Eliminating the ability to store first class procedures we get:

```
frilled.cs.indiana.edu%cat calc
#!/usr/bin/perl

$acc = $ARGV[0] + $ARGV[1];

print "The accumulator is now: $acc\n$0 $acc <new-arg>\n";
frilled.cs.indiana.edu%
```

Here's a sample session:

```
frilled.cs.indiana.edu%./calc 10 0
The accumulator is now: 10
./calc 10 <new-arg>
frilled.cs.indiana.edu%./calc 10 -5
The accumulator is now: 5
./calc 5 <new-arg>
frilled.cs.indiana.edu%./calc 5 8
The accumulator is now: 13
./calc 13 <new-arg>
frilled.cs.indiana.edu%
```

[3]There's something specific about this pattern: note that it *steps through* the underlying stages of recursion. The web is the ultimate example of a lazy evaluation context.

But the good news is that the browser can simulate the storage of first class procedures because it can encapsulate all the information required for the next call under a hyperlink.

Our Basic Pattern

There are six steps to this pattern:

1. retrieve state
2. if (state is empty)
   (a) initialize state
3. else
   (a) read input
   (b) update state (using input)
4. store state
5. show state
6. get ready for more input

Testing and Debugging

First build a program in the language of your choice. No network is needed. Test and debug as much as you want. Wrap the basic pattern in an infinite loop. Here's what the program above becomes[4]:

```
burrowww.cs.indiana.edu% cat calc
#!/usr/bin/perl
$acc = "";
$msg = "Welcome.";
while (true) {
  # retrieve state (no need to: $acc)
  if ($acc eq "") { # if state is empty
    $acc = 0; # initialize state
  } else { # else read input and process state
    $arg = <STDIN>; # read input
    $acc += $arg; # update state (using input)
    $msg = "The acumulator is currently: " . $acc;
  }
  # store state (no need to: $msg, $acc)
  print $msg . "\n"; # show state
  print "Add this: ";
}
burrowww.cs.indiana.edu% ./calc
Welcome.
Add this: 10
The acumulator is currently: 10
```

[4]Comments, as well as computer output, are shown in lightgrey.

```
Add this: -3
The acumulator is currently: 7
Add this: 4
The acumulator is currently: 11
Add this:
The acumulator is currently: 11
Add this:
The acumulator is currently: 11
Add this: 10
The acumulator is currently: 21
Add this: ^Cburrowww.cs.indiana.edu%
```

This is now easy to convert[5] into a CGI script, as shown below. Notice that each of the preceding stages are accounted for and that the `while` loop disappears completely[6].

```
#!/usr/bin/perl
use CGI; # import a package
$q = new CGI; # read input (very general)
$acc = $q->param('acc'); # retrieve state (client-side)
if ($acc eq "") { # state is empty
  $acc = 0; # initialize state
  $msg = "Welcome. <p> The accumulator is currently: 0";
} else { # process state
  $arg = $q->param('arg'); # read input (very specific)
  $acc += $arg; # update state (using input)
  $msg = "The accumulator is currently: " . $acc;
}

print $q->header, $q->start_html,
      qq{<form action="}, $q->script_name, qq{">},
      qq{<input type="hidden"
                name="acc" value="$acc">}, # store state
      $msg, "<p>", # show state
      qq{Add this:
         <input type="text"
                name="arg">}, # prepare for more input
      qq{</form>},
      $q->end_html;
```

Examples

We now present examples in CGI/Perl, PHP, Javascript, Java servlets and Java Server Pages. As you'll see the pattern applies equally well regardless of where we decide to keep state, client or server[7].

[5] Have you noticed the two empty input lines above?

[6] The loop only happens on one computer. Since the client now is in a browser on the web, keeping the loop would literally exclude the (remote) client.

[7] The example above relies exclusively on the client browser, but other approaches exist.

A Vending Machine

Let's implement a vending machine. The interface contains a `select` field that lets the user choose the name of a coin, but sends a numeric value to the server. The listed coins will be: dollar (with a numeric value of 100), quarter (25), dime (10), nickel (5) and cent (1).

The machine sells stamps by the book and the price of a book of stamps is $3.70. The machine reports your current credit and should notify you when a book of stamps will be dispensed for you. It should also notify you of any change you need to receive, when that happens. Let's start by writing a prototype (say, in Java).

The Prototype

```
burrowww.cs.indiana.edu% pwd
/nfs/paca/san/r1a011/dgerman/final
burrowww.cs.indiana.edu% cat One.java
class One {
    public static void main(String[] args) throws java.io.IOException {
        String credit = null; // your state (part one)
        String message = "Welcome."; // your state (part two)
        while (true) {
            if (credit == null) { // retrieve state
                credit = "0"; // initialize if inexistent (empty)
            } else { // process state if state can be retrieved
                int coin = Integer.parseInt( // read input
                               (new java.io.BufferedReader(
                                   new java.io.InputStreamReader
                                       (System.in))).readLine());
                credit = (Integer.parseInt(credit) + coin) + "";
                if (Integer.parseInt(credit) < 370) {
                    message = "Please enter more coins.";
                } else {
                    message = "The stamps are yours. \nChange: " +
                              (Integer.parseInt(credit) - 370) +
                              " (cents).\n" + "Welcome.";
                    credit = "0";
                }
            } // end of process state (using input to update state)
            // no need to save state in the prototype (credit, message)
            System.out.println(message); // report state (part two)
            System.out.println("Your credit is: " +
                               credit); // report state (part one)
            System.out.print("Enter coins here: ");
                            // get ready for more input...
        }
    }
}
burrowww.cs.indiana.edu% javac One.java
```

```
burrowww.cs.indiana.edu% java One
Welcome.
Your credit is: 0
Enter coins here: 340
Please enter more coins.
Your credit is: 340
Enter coins here: 45
The stamps are yours.
Change: 15 (cents).
Welcome.
Your credit is: 0
Enter coins here: 20
Please enter more coins.
Your credit is: 20
Enter coins here: 351
The stamps are yours.
Change: 1 (cents).
Welcome.
Your credit is: 0
Enter coins here: 370
The stamps are yours.
Change: 0 (cents).
Welcome.
Your credit is: 0
Enter coins here: 10
Please enter more coins.
Your credit is: 10
Enter coins here: 1
Please enter more coins.
Your credit is: 11
Enter coins here: 346
Please enter more coins.
Your credit is: 357
Enter coins here: 30
The stamps are yours.
Change: 17 (cents).
Welcome.
Your credit is: 0
Enter coins here: ^Cburrowww.cs.indiana.edu%
```

As before, comments and computer output are shown in `lightgrey`.

Let's see now how easy it is to transform this code into a web-based implementation.

CGI

Assume we keep state on the client-side[8] in `hidden` fields.

[8] Keeping state on the server side is more involved because we need to set up the tables in the database (so we start with the simpler case).

Then we have the following immediate translation.

```
#!/usr/bin/perl

use CGI; # import a package

$q = new CGI; # read input (very general)

$msg = "Welcome."; # almost part of the state
$credit = $q->param('credit'); # retrieve state

if ($credit eq "") { # if state is empty
  $credit = "0"; # initialize state

} else { # if valid state has been retrieved

  $coin = $q->param('coin'); # read input (very specific)

  $credit += $coin; # update state (using input)

  if ($credit < 370) { # in which we implement reset
    $msg = "Please enter more coins.";
  } else {
    $msg = "The stamps are yours.<br>Change: " .
           ($credit - 370) . "cents.<br>Welcome.";
    $credit = 0;
  }
} # end of update state using input

# store state, show state, get ready for more input
print $q->header, $q->start_html, qq{
  <form action="}, $q->script_name, qq{">
    <input type="hidden" name="credit" value="$credit">
    $msg <p>
    Your credit is: $credit <p>
    Enter coins here: <select name="coin">
      <option value="nothing"> Click me!
      <option value="1"> Cent
      <option value="5"> Nickel
      <option value="10"> Dime
      <option value="25"> Quarter
      <option value="100"> Dollar
    </select> <p>
    Then choose <input type="submit" value="Proceed">
  </form>
}, $q->end_html;
```

We now move to PHP with hidden fields.

PHP

Comments but also expressions are in `lightgrey` below.

```
<? $message = "Welcome."; // almost part of state
   // retrieve state ($state is immediately available)
   if ($credit == "") { // if state is empty
      $credit = "0"; // initialize state
   } else { // otherwise update state (using input)
      // $coin is immediately available (read input)
      $credit += $coin;
      if ($credit < 370) { // in which we implement reset
         $msg = "Please enter more coins.";
      } else {
         $msg = "The stamps are yours.<br>Change: " .
                ($credit - 370) . " cents.<br>Welcome.";
         $credit = 0; // reset
      }
   }

 ?>

<html>
  <head>
    <title>Vending Machine</title>
  </head>
  <body bgcolor="white">
    <form action="<?=$SCRIPT_NAME?>">
      <input type="hidden" name="credit" value="<?=$credit?>">
      <?=$msg?> <p>
      Your credit is: <?=$credit?> <p>
      Enter coins here: <select name="coin">
        <option value="nothing"> Click Me!
        <option value="1"> Cent
        <option value="5"> Nickel
        <option value="10"> Dime
        <option value="25"> Quarter
        <option value="100"> Dollar
      </select> <p>
      Then choose <input type="submit" value="Proceed">
    </form>
  </body>
</html>
```

I think we should leave the server-side state maintenance techniques for a second part of this review. So let's move on to JavaScript and then implement the same thing using Java servlets and Java server pages using `hidden` fields to store state information.

JavaScript

Here the emphasis on event-driven programming is more noticeable.

```
<html>
  <head>
    <title>Vending Machine</title>
    <script language="javascript">
      var credit, msg = "Welcome";
      function processRequest() { // update state (using input)
        var ind = document.forms[0].coin.selectedIndex; // read input
        var coin = document.forms[0].coin.options[ind].value;
        if (credit == undefined) { // empty state
          credit = 0; // initialize it
        } else { // otherwise update it
          credit += eval(coin); // based on input
          if (credit < 370) { // implement reset
            msg = "Please enter more coins.";
          } else {
            msg = "The stamps are yours.<br>Change:" +
                  (credit - 370) + " cents.<br>Welcome.";
            credit = 0;
          }
        } // no need to store state
        document.getElementById("show").innerHTML = msg + // show state
          "<p>" + "Your credit is: " + credit + " cents.";
      } // we're always ready for more input (see below)
    </script>
  </head>
  <body bgcolor="white">
    <form>
      <div id="show">
        Welcome. <p>
        Your credit is: 0 cents.
      </div>
      Enter coins here: <select name="coin">
        <option value="nothing"> Click Me!
        <option value="1"> Cent
        <option value="5"> Nickel
        <option value="10"> Dime
        <option value="25"> Quarter
        <option value="100"> Dollar
      </select> <p>
      Then click <input type="button"
                        value="Proceed"
                        onClick="processRequest()">
    </form>
  </body>
</html>
```

The next two stages should resemble CGI (servlets) and PHP (JSP).

Java Servlets

Fortunately the prototype is already in Java.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*; // packages to import
public class Vending extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
              throws ServletException, IOException {
      String credit, message = "Welcome."; // your state
      credit = request.getParameter("credit"); // retrieve state
      if (credit == null) { // empty state?
          credit = "0"; // initialize it
      } else { // otherwise update (using input)
          int coin = Integer.parseInt( // get input (specific)
                        request.getParameter("coin"));
          credit = (Integer.parseInt(credit) + coin) + ""; // update
          if (Integer.parseInt(credit) < 370) { // in case of reset
              message = "Please enter more coins.";
          } else {
              message = "The stamps are yours. <br>Change: " +
                        (Integer.parseInt(credit) - 370) +
                        " (cents).<br>" + "Welcome.";
              credit = "0";
          }
      } // end of processing state
      response.setContentType("text/html"); // get ready to print
      PrintWriter out = response.getWriter();
      out.println("<html><head><title>Vending Machine</title></head>" +
"<body bgcolor=\"white\">" + "<form action=\"" +
    request.getContextPath() + request.getServletPath() + "\">" +
// store state (below)
"<input type=\"hidden\" name=\"credit\" value=\"" + credit + "\">" +
// show state
message + "<p>" + " Your credit is: " + credit + "<p>" +
// get ready for more input
"Enter coins here: <select name=\"coin\">" +
"  <option value=\"nothing\"> Click me! " +
"  <option value=\"1\"> Cent " +
"  <option value=\"5\"> Nickel " +
"  <option value=\"10\"> Dime " +
"  <option value=\"25\"> Quarter " +
"  <option value=\"100\"> Dollar " +
"</select> <p> Then click <input type=\"submit\" value=\"Proceed\">" +
"</form></body></html>" );
  }
}
```

Quick question: how do you fix the "no input" submission?

Java Server Pages

We now experience a few simplifications.

```
<%    String credit, message = "Welcome."; // your state
      credit = request.getParameter("credit"); // retrieve state
      if (credit == null) { // empty state?
          credit = "0"; // initialize it
      } else { // otherwise update (using input)
          int coin = Integer.parseInt( // get input (specific)
                        request.getParameter("coin"));
          credit = (Integer.parseInt(credit) + coin) + ""; // update
          if (Integer.parseInt(credit) < 370) { // in case of reset
              message = "Please enter more coins.";
          } else {
              message = "The stamps are yours. <br>Change: " +
                        (Integer.parseInt(credit) - 370) +
                        " (cents).<br>" + "Welcome.";
              credit = "0";
          }
      } // end of processing state
 %>

<html>
  <head>
    <title>Vending Machine</title>
  </head>
  <body bgcolor="white">
    <form
      action="<%=request.getContextPath() + request.getServletPath()%>">
      <input type="hidden" name="credit" value="<%=credit%>">
      <%=message%> <p> Your credit is: <%=credit%> <p>
      Enter coins here: <select name="coin">
        <option value="nothing"> Click me!
        <option value="1"> Cent
        <option value="5"> Nickel
        <option value="10"> Dime
        <option value="25"> Quarter
        <option value="100"> Dollar
      </select>
      <p> Then click <input type="submit" value="Proceed">
    </form>
  </body>
</html>
```

Conclusion

In spite of still having to discuss the implementations that keep state on the server side, just knowing this for the exam would be enough I think.