Managing state for a “stateless” world.

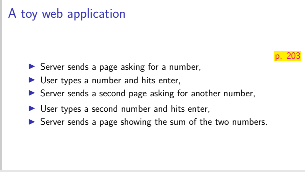
David Bremner

March 11, 2024

A toy web application

p. 203

- Server sends a page asking for a number,
- User types a number and hits enter,
- Server sends a second page asking for another number,
- User types a second number and hits enter,
- Server sends a page showing the sum of the two numbers.



A toy web application

- Server sends a page asking for a number,
- User types a number and hits enter,
- Server sends a second page asking for another number,
- User types a second number and hits enter,
- Server sends a page showing the sum of the two numbers.

1. The HTTP protocol is *stateless*: each HTTP query can be thought of as running a program (or a function), getting a result, then killing it. This makes interactive applications hard to write.
   The basic problem here is HTTP’s statelessness, something that both web servers and web browsers use extensively. Browsers give you navigation buttons and sometimes will not even communicate with the web server when you use them (instead, they’ll show you cached pages), they give you the ability to open multiple windows or tabs from the current one, and they allow you to ”clone” the current tab. If you view each set of HTTP queries as a session – this means that web browsers allow you to go back and forth in time, explore multiple futures in parallel, and clone your current world.
2. You can obviously ask for both numbers on the same page, but the interleaved compution style is common.

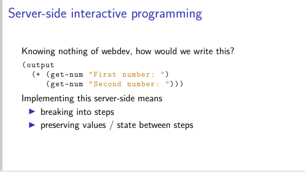
Server-side interactive programming

Knowing nothing of webdev, how would we write this?

```
(output
  (+ (get-num "First number: ")
     (get-num "Second number: ")))
```

Implementing this server-side means

- breaking into steps
- preserving values / state between steps



Server-side interactive programming

Knowing nothing of webdev, how would we write this?

```
(output
  (+ (get-num "First number: ")
     (get-num "Second number: ")))
```

Implementing this server-side means

- breaking into steps
- preserving values / state between steps

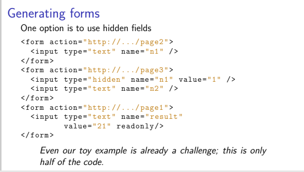
1. The interaction is limited to presenting the user with some data and that’s all – you cannot do any kind of interactive querying. We therefore must turn this server function into three separate functions: one that shows the prompt for the first number, one that gets the value entered and shows the second prompt, and a third that shows the results page.

Generating forms

One option is to use hidden fields

```
<form action="http://.../page2">
  <input type="text" name="n1" />
</form>
<form action="http://.../page3">
  <input type="hidden" name="n1" value="1" />
  <input type="text" name="n2" />
</form>
<form action="http://.../page1">
  <input type="text" name="result"
         value="21" readonly/>
</form>
```

Even our toy example is already a challenge; this is only half of the code.



Generating forms

One option is to use hidden fields

```
<form action="http://.../page2">
  <input type="text" name="n1" />
</form>
<form action="http://.../page3">
  <input type="hidden" name="n1" value="1" />
  <input type="text" name="n2" />
</form>
<form action="http://.../page1">
  <input type="text" name="result"
        value="21" readonly/>
</form>
```

Even our toy example is already a challenge; this is only half of the code.

1. The state is all saved in the client browser – if it dies, then the interaction is gone.
2. The state might even include values that are not expressible as part of the form (for example an open database connection or a running process).
3. Another common approach is to store the state information on the server, and use a small handle (eg, in a cookie) to identify the state, then each function can use the cookie to retrieve the current state of the service – but this is exactly how we get to the above bugs. It will fail with any of the mentioned time-manipulation features.

Reintroduction to Continuations: Web Programming

Re-start with the original (idealized) expression:

p. 204

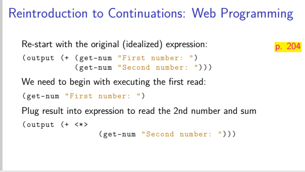
```
(output (+ (get-num "First number: ")
           (get-num "Second number: ")))
```

We need to begin with executing the first read:

```
(get-num "First number: ")
```

Plug result into expression to read the 2nd number and sum

```
(output (+ <*>
                (get-num "Second number: ")))
```



Reintroduction to Continuations: Web Programming

Re-start with the original (idealized) expression: p. 204

```
(output (+ (get-num "First number: ")
           (get-num "Second number: ")))
```

We need to begin with executing the first read:

```
(get-num "First number: ")
```

Plug result into expression to read the 2nd number and sum

```
(output (+ <*>
           (get-num "Second number: ")))
```

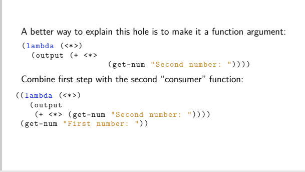
1. Assuming for now that ‘get-num’ is implemented and works
2. That’s the same as the first expression, except that instead of the first ‘get-num’ we use a ”hole”:
3. ‘⟨*⟩’ marks the point where we need to plug the result of the first question into.

A better way to explain this hole is to make it a function argument:

```
(lambda (<*>)
  (output (+ <*>
                  (get-num "Second number: "))))
```

Combine first step with the second "consumer" function:

```
((lambda (<*>)
   (output
    (+ <*> (get-num "Second number: "))))
 (get-num "First number: "))
```



A better way to explain this hole is to make it a function argument:

```
(lambda (<*>)
  (output (+ <*>
                (get-num "Second number: "))))
```

Combine first step with the second “consumer” function:

```
((lambda (<*>)
   (output
    (+ <*> (get-num "Second number: "))))
 (get-num "First number: "))
```

1. Actually, we can split the second and third steps in the same way.

Continue by splitting the body of the consumer:

```
(output (+ <*> (get-num "Second number: ")))
```

Into a “reader” and the rest of the computation (using a new hole):

```
(get-num "Second number: ")    ; reader part

     (output (+ <*> <*2>)) ; rest of comp
```

Doing all of this gives us:

getnum1
```
((lambda (<*1>)
   ((lambda (<*2>)
      (output (+ <*1> <*2>)))
    (get-num "Second number: ")))
 (get-num "First number: "))
```

- This works, but is not much fun to read

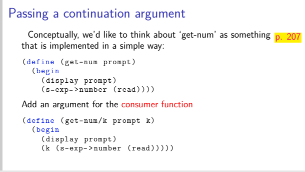
Passing a continuation argument

Conceptually, we'd like to think about 'get-num' as something that is implemented in a simple way: p. 207

```
(define (get-num prompt)
  (begin
    (display prompt)
    (s-exp->number (read))))
```

Add an argument for the consumer function

```
(define (get-num/k prompt k)
  (begin
    (display prompt)
    (k (s-exp->number (read)))))
```



Passing a continuation argument

Conceptually, we’d like to think about ‘get-num’ as something that is implemented in a simple way: p. 207

```
(define (get-num prompt)
  (begin
    (display prompt)
    (s-exp->number (read))))
```

Add an argument for the consumer function

```
(define (get-num/k prompt k)
  (begin
    (display prompt)
    (k (s-exp->number (read)))))
```

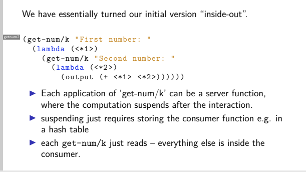
1. In this version of ‘get-num’ the ‘k’ argument is the **continuation** of the computation. (‘k’ is a common name for a continuation argument.)

We have essentially turned our initial version “inside-out”.

getnum2
```
(get-num/k "First number: "
  (lambda (<*1>)
    (get-num/k "Second number: "
      (lambda (<*2>)
        (output (+ <*1> <*2>))))))
```

- Each application of ‘get-num/k’ can be a server function, where the computation suspends after the interaction.
- suspending just requires storing the consumer function e.g. in a hash table
- each `get-num/k` just reads – everything else is inside the consumer.

Managing state for a “stateless” world.
└─From Web Apps to Continuations


We have essentially turned our initial version “inside-out”.

getnum2
```
(get-num/k "First number: "
  (lambda (<*1>)
    (get-num/k "Second number: "
      (lambda (<*2>)
        (output (+ <*1> <*2>))))))
```

- Each application of ‘get-num/k’ can be a server function, where the computation suspends after the interaction.
- suspending just requires storing the consumer function e.g. in a hash table
- each `get-num/k` just reads – everything else is inside the consumer.

1. This means that the ”submit” button will somehow encode a reference to the hash table that can make the next service call retrieve the stored function.

Trivial continuations

getnum3
```
(output
 (+ (get-num/k "First number: " (lambda (<*>) <*>))
    (get-num/k "Second number: " (lambda (<*>) <*>))))
```

- this lacks sequencing
- it works OK here, but will be problematic “on the web”

Simulating web reading

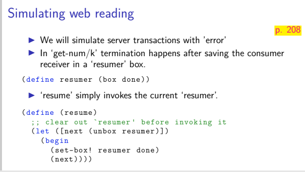
p. 208

- We will simulate server transactions with 'error'
- In 'get-num/k' termination happens after saving the consumer receiver in a 'resumer' box.

```
(define resumer (box done))
```

- 'resume' simply invokes the current 'resumer'.

```
(define (resume)
  ;; clear out `resumer' before invoking it
  (let ([next (unbox resumer)])
    (begin
      (set-box! resumer done)
      (next))))
```



Simulating web reading

p. 208

- We will simulate server transactions with ’error’
- In ‘get-num/k’ termination happens after saving the consumer receiver in a ‘resumer’ box.

```
(define resumer (box done))
```

- ‘resume’ simply invokes the current ‘resumer’.

```
(define (resume)
  ;; clear out `resumer' before invoking it
  (let ([next (unbox resumer)])
    (begin
      (set-box! resumer done)
      (next))))
```

1. The book uses stacker here instead of DrRacket; you may want to try working through those (short) examples on your own.
2. Instead of storing just the receiver there, we will store a function that does the prompting and the reading and then invoke the receiver. ‘resumer’ is therefore bound to a box that holds a no-argument function that does the work of resuming the computation, and when there is nothing next, it is bound to a ‘done’ function that throws an appropriate error.

Yielding via error

- We want to convince ourselves the execution can be stopped completely and resumed, so we use `error`

```
(define (done)
  (error 'resume "nothing suspended."))

(define (yield/k prompt k)
  (begin (set-box! resumer k) (error 'yield
     prompt)))

(define (output n) (yield/k (to-string n) done))
```

Simulated Web Transactions with Yield and Resume

fakeweb

```
(define (get-num/k prompt k)
  (yield/k  "(resume) to go"
            (lambda () (begin (display prompt)
                              (k (s-exp->number
                                  (read))))))))

(define (example)
  (get-num/k "First number: "
             (lambda (n1)
               (get-num/k "Second number: "
                          (lambda (n2)
                            (output (+ n1 n2)))))))
```

You can also try the bogus expression that we mentioned:

getnum4
```
(define (example2)
  (output
   (+ (get-num/k "First number: " (lambda (n) n))
      (get-num/k "Second number: " (lambda (n) n)))))
```

and see how it breaks.

Yield revisited

p. 209

- Recall that `(let/cc id ex)` binds id to the "current-continuation"
- Yield is almost the same, but the continuation argument is implicit

```
(define (yield prompt)
  (let/cc k
    (begin (set-box! resumer k)
           (error 'yield prompt))))
```

Resume revisited

- Can you spot the minor change to `resume`?

```
(define (resume)
  (let ([next (unbox resumer)])
    (begin
      (set-box! resumer done)
      (next 'dummy))))
```

Continuations are hidden from “user code”

fweb-cc2

```
(define (output n) (error 'output (to-string n)))

(define (get-num prompt)
  (begin
    (yield "(resume) to continue")
    (display prompt)
    (s-exp->number (read))))

(define (example)
  (output
   (+ (get-num "First number: ")
      (get-num "Second number: "))))
```

It's easy to convert our example to a real "web app" using Racket's web server framework, and the core of the code looks very simple:

webapp
```
(define (start initial-request)
  (page "The sum is: "
        (+ (get-num "First number: ")
           (get-num "Second number: "))))
```

Page is a simple HTML generation function

webapp
```
(define (page . lst)
  (response/xexpr
   `(html (body ,@(map
                   (lambda(x) (if (number? x)(~a x) x))
                   lst)))))
```

The core of `get-num` is written in continuation passing style

webapp

```
(define (get-num-core prompt)
  (lambda (k)
    (page
     `(form ([action ,k])
            ,prompt
            (input ([type "text"][name "n"]))))))
```

send/suspend captures the calling context, and allows resuming by visiting a URL

webapp

```
(define (get-num prompt)
  (string->number
   (extract-binding/single
    'n
    (request-bindings
     (send/suspend (get-num-core prompt))))))
```