A toy web application

[[[PLAI Chapter 14]]]

- 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.

2019-03-29

A toy web application

[[[PLAI Chapter 14]]]

- 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. Obviously, writing programs to run on a web server is a profitable activity, and therefore highly desirable. But when we do so, we need to somehow cope with the web's statelessness. To see the implications from a PL point of view we'll use a small "toy" example that demonstrates the basic issues – an "addition" service:

re problem.

Starting from just that, consider how you'd **want** to write the code for such a service. (If you have experience writing web apps, then try to forget it for now.)

```
(web-display
  (+ (web-read "First number")
     (web-read "Second number")))
```

But this is not going to work, because prompting and reading are separate operations in web apps.

Starting from just that, consider how you'd **want** to write the code for such a service. (If you have experience writing web apps, then try to forget it for now.)

1
```
(web-display
  (+ (web-read "First number")
     (web-read "Second number")))
```

But this is not going to work, because prompting and reading are separate operations in web apps.

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.

Assuming a generic ”query argument” that represents the browser request, and a return value that represents a page for the browser to render, we have:

2

```
(define (page1 query)
  ... show the first question ...)

(define (page2 query)
  ... extract the number from the query ...
  ... show the second question ...)

(define (page3 query)
  ... extract the number from the query ...
  ... show the sum ...)
```

- Note that ‘page2’ receives the first number directly, but ‘page3’ doesn’t.
- A typical hack to get around this is to use a "hidden field" in the HTML form that ‘page2’ generates, where that field holds the second result.

To make things more concrete, we'll use some imaginary web API functions:

3

```
(define (page1 query)
  (web-read "First number" 'n1 "page2"))

(define (page2 query)
  (let ([n1 (get-field query 'n1)])
    (with-hidden-field 'n1 n1
      (web-read "Second number" 'n2 "page3"))))

(define (page3 query)
  (web-display
    "Your two numbers sum up to: "
    (+ (get-field query 'n1)
       (get-field query 'n2))))
```

Which would (supposedly) result in something like the following html forms when the user enters 1 and 20:

```
<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>
```

- gets very difficult to manage with real services where the ”state” of the server is more complex
- the state is all saved in the client browser – if it dies, then the interaction is gone.

- gets very difficult to manage with real services where the "state" of the server is more complex
- the state is all saved in the client browser – if it dies, then the interaction is gone.

1. and it might even include values that are not expressible as part of the form (for example an open database connection or a running process).
2. 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.

Introduction to Continuations: Web Programming

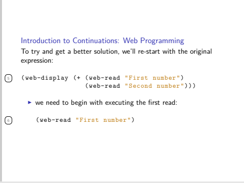
To try and get a better solution, we'll re-start with the original expression:

```
(web-display (+ (web-read "First number")
                (web-read "Second number")))
```

- we need to begin with executing the first read:

```
    (web-read "First number")
```

2019-03-29



Introduction to Continuations: Web Programming

To try and get a better solution, we'll re-start with the original expression:

```
(web-display (+ (web-read "First number")
                (web-read "Second number")))
```

- we need to begin with executing the first read:

```
   (web-read "First number")
```

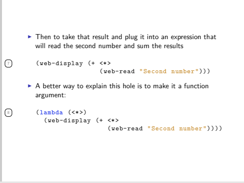
1. assuming that 'web-read' works

- Then to take that result and plug it into an expression that will read the second number and sum the results

7
```
(web-display (+ <*>
                (web-read "Second number")))
```

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

8
```
(lambda (<*>)
  (web-display (+ <*>
                  (web-read "Second number"))))
```



- Then to take that result and plug it into an expression that will read the second number and sum the results

7
```
(web-display (+ <*>
                (web-read "Second number")))
```

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

8
```
(lambda (<*>)
  (web-display (+ <*>
                  (web-read "Second number"))))
```

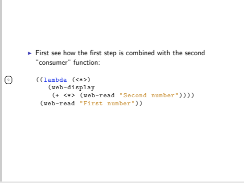
1. that's the same as the first expression, except that instead of the first 'web-read' we use a "hole":
2. '⟨*⟩' marks the point where we need to plug the result of the first question into.

- First see how the first step is combined with the second ”consumer” function:

9

```
((lambda (<*>)
   (web-display
    (+ <*> (web-read "Second number"))))
 (web-read "First number"))
```

2019-03-29



- First see how the first step is combined with the second "consumer" function:

9

```
((lambda (<*>)
   (web-display
    (+ <*> (web-read "Second number"))))
 (web-read "First number"))
```

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

- And continue by splitting the body of the consumer:

10
```
(web-display (+ <*> (web-read "Second
   number")))
```

into a "reader" and the rest of the computation (using a new hole):

11
```
(web-read "Second number")    ; reader part

     (web-display (+ <*> <*2>))
                               ; rest of comp
```

Doing all of this gives us:

12

```
((lambda (<*1>)
   ((lambda (<*2>)
      (web-display (+ <*1> <*2>)))
    (web-read "Second number")))
 (web-read "First number"))
```

And now we can proceed to the main trick. Conceptually, we'd like to think about 'web-read' as something that is implemented in a simple way:

13

```
(define (web-read prompt)
  (printf "~a: " prompt)
  (read-number))
```

To accommodate the above ”hole functions”, we change it by adding an argument for the consumer function that we need to pass the result to, and call it ‘web-read/k’:

14

```
(define (web-read/k prompt k)
  (display prompt)
  (k (read-number)))
```

To accommodate the above "hole functions", we change it by adding an argument for the consumer function that we need to pass the result to, and call it 'web-read/k':

14
```
(define (web-read/k prompt k)
  (display prompt)
  (k (read-number)))
```

1. In this version of 'web-read' the 'k' argument is the **continuation** of the computation. ('k' is a common name for a continuation argument.)

This is not too different from the previous version – the only difference is that we make the function take a consumer function as an input, and hand it what we read instead of just returning it.

15

```
(web-read/k "First number"
  (lambda (<*1>)
    (web-read/k "Second number"
      (lambda (<*2>)
        (web-display (+ <*1> <*2>))))))
```

This looks a bit complicated...

2019-03-29

This is not too different from the previous version – the only difference is that we make the function take a consumer function as an input, and hand it what we read instead of just returning it.

15

```
(web-read/k "First number"
  (lambda (<*1>)
    (web-read/k "Second number"
      (lambda (<*2>)
        (web-display (+ <*1> <*2>))))))
```

This looks a bit complicated...

1. Using it makes things a little easier, since we pass the hole function which gets called automatically, instead of combining things ourselves.

In our simulated environment, it seems we should get exactly the same result with

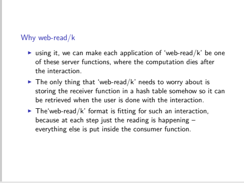
16
```
(web-display
 (+ (web-read/k "First number" (lambda (<*>) <*>))
    (web-read/k "Second number" (lambda (<*>)
       <*>))))
```

but then there's not much point to having 'web-read/k' at all...

Why web-read/k

- using it, we can make each application of ‘web-read/k’ be one of these server functions, where the computation dies after the interaction.
- The only thing that ‘web-read/k’ needs to worry about is storing the receiver function in a hash table somehow so it can be retrieved when the user is done with the interaction.
- The‘web-read/k’ format is fitting for such an interaction, because at each step just the reading is happening – everything else is put inside the consumer function.

2019-03-29



Why web-read/k

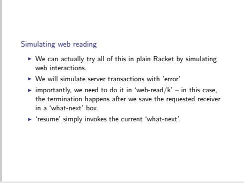
- using it, we can make each application of ‘web-read/k’ be one of these server functions, where the computation dies after the interaction.
- The only thing that ‘web-read/k’ needs to worry about is storing the receiver function in a hash table somehow so it can be retrieved when the user is done with the interaction.
- The‘web-read/k’ format is fitting for such an interaction, because at each step just the reading is happening – everything else is put inside the consumer function.

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.

Simulating web reading

- We can actually try all of this in plain Racket by simulating web interactions.
- We will simulate server transactions with 'error'
- importantly, we need to do it in ‘web-read/k’ – in this case, the termination happens after we save the requested receiver in a ‘what-next’ box.
- ‘resume’ simply invokes the current ‘what-next’.

2019-03-29



Simulating web reading

- We can actually try all of this in plain Racket by simulating web interactions.
- We will simulate server transactions with 'error'
- importantly, we need to do it in 'web-read/k' – in this case, the termination happens after we save the requested receiver in a 'what-next' box.
- 'resume' simply invokes the current 'what-next'.

1. This is useful to look at the core problem while avoiding the whole web mess that is uninteresting for this discussion.
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. 'what-next' 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 'nothing-to-do' function that throws an appropriate error.

Simulated Web Framework 1/2

17

```
(define (nothing-to-do) (error "nothing
    suspended."))
(define what-next (box nothing-to-do))

(define (web-display n)
  (set-box! what-next nothing-to-do)
  (error 'web-output "~s" n))

(define (web-read/k prompt k)
  (set-box! what-next
       (lambda () (printf "~a: " prompt) (k
          (read))))
  (error 'web-read/k "enter (resume) to continue"))
```

18

```
(define (resume)
  ;; to avoid mistakes, we clear out `what-next'
     before invoking it
  (let ([next (unbox what-next)])
    (set-box! what-next nothing-to-do)
    (next)))

(define (example)
  (web-read/k
   "First number"
   (lambda (n1)
     (web-read/k
      "Second number"
      (lambda (n2)
        (web-display (+ n1 n2)))))))
```

You can also try the bogus expression that we mentioned:

19

```
(define (example2)
  (web-display
   (+ (web-read/k "First number" (lambda (n) n))
      (web-read/k "Second number" (lambda (n)
         n))))))
```

and see how it breaks.

- It's easy to write this addition server using Racket's web server framework, and the core of the code looks very simple:

```
(define (start initial-request)
  (page "The sum is: "
        (+ (web-read "First number")
           (web-read "Second number"))))
```

```
#lang web-server/insta
(define (page . lst)
  (response/xexpr
   `(html
     (body ,@(map (λ(x) (if (number? x)(~a x) x))
                  lst)))))
(define (web-read prompt)
  (string->number
   (extract-binding/single
    'n
    (request-bindings
     (send/suspend
      (λ (k)
        (page
         `(form ([action ,k])
           ,prompt ": " (input ([type "text"]
                                [name
                                   "n"]))))))))))

(define (start initial-request)
  (page "The sum is: "
        (+ (web-read "First number: ")
```

- `page` and `web-read` are just generating html.
- The main piece of magic there is in ‘send/suspend’ which makes the web server capture the computation’s continuation and store it in a hash table, to be retrieved when the user visits the given URL.