PL: Lecture #27

Tuesday, December 3rd

Web Programming

> 👉 ***PLAI §15***

Consider web programming as a demonstration of a frequent problem. 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.

> *For example, consider this behavior (which is based on a real story of a probably not-so-real bug known as "the ITA bug"):*
>
> - *You go on a flight reservation website, and look at flights to Paris or London for a vacation.*
> - *You get a list of options, and choose one for Paris and one for London, ctrl-click the first and then the second to open them in new tabs.*
> - *You look at the descriptions and decide that you like the first one best, so you click the button to buy the ticket.*
> - *A month later you go on your plane, and when you land you realize that you're in the wrong country — the ticket you paid for was the second one after all…*

Obviously there is some fundamental problem here — especially given that this problem plagued many websites early on (and these days these kind of problems can still be found in some places (like the registrar's system), except that people are much more aware of it, and are much more prepared to deal with it). In an attempt to clarify what it is exactly that went wrong, we might require that each interaction will result in something that is deterministically based on what the browser window shows when the interaction is made — but even that is not always true. Consider the same scenario except with a bookstore and an "add to my cart" button. In this case you *want* to be able to add one item to the cart in the first window, then switch to the second window and click "add" there too: you want to end up with a cart that has *both* items.

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.

These are features that the HTTP protocol intentionally allows by being stateless, and that people have learned to use effectively. A stateful protocol (like ssh, or ftp) will run in a single process (or a program, or a function) that is interacting with you directly, and this process dies only when you disconnect. A big advantage of stateful protocols is their ability to be very interactive and rely on state (eg, an editor updates a character on the screen, relying on the rest of it showing the same text); but stateless protocols can scale up better, and deal with a more hectic kind of interaction (eg, open a page on an online store, keep it open and buy the item a month later; or any of the above "time manipulation" devices).

> *Side-note: Some people think that Ajax is the answer to all of these problems. In reality, Ajax is layered on top of (asynchronous) web queries, so in fact it is the exact same situation. You do have an option of creating an application that works completely on the client side, but that wouldn't be as attractive — and even if you do so, you're still working inside a browser that can play the same time tricks.*

Basic web programming

> 👉 ***PLAI §16***

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:

- 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.

[Such a small application is not realistic, of course: you can obviously ask for both numbers on the same page. We still use it, though, to minimize the general interaction problem to a more comprehensible core 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 all of that now, and focus on just this basic problem.) The plain version of what we want to implement is:

```
(print
  (+ (read "First number")
     (read "Second number")))
```

which is trivially "translated" to:

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

But this is never going to work. The interaction is limited to presenting the user with some data and that's all — you cannot do any kind of interactive querying. For the purpose of making this more concrete, imagine that `web-read` and `web-display` both communicate information to the user via something like `error`: the information is sent and at the same time the whole computation is aborted. With this, the above code will just manage to ask for the first number and nothing else happens.

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. The first two of these functions would send the information (and the respective computation dies) to the browser, including a form submission URL that will invoke the next function.

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:

```
(define (f1 query)
  ... show the first question ...)

(define (f2 query)
  ... extract the number from the query ...
  ... show the second question ...)

(define (f3 query)
  ... extract the number from the query ...
  ... show the sum ...)
```

Note that `f2` receives the first number directly, but `f3` doesn't. Yet, it is obviously needed to show the sum. A typical hack to get around this is to use a "hidden field" in the HTML form that `f2` generates, where that field holds the second result. To make things more concrete, we'll use some imaginary web API functions:

```
(define (f1 query)
  (web-read "First number" 'n1 "f2"))

(define (f2 query)
```

```
  (let ([n1 (get-field query 'n1)])
    ;; imagine that the following "configures" what web-read
    ;; produces by adding a hidden field to display
    (with-hidden-field 'n1 n1
      (web-read "Second number" 'n2 "f3"))))

(define (f3 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 2:

```
http://.../f1
<form action="http://.../f2">
  First number:
  <input type="text" name="n1" />
</form>

http://.../f2
<form action="http://.../f3">
  <input type="hidden" name="n1" value="1" />
  Second number:
  <input type="text" name="n2" />
</form>

http://.../f3
<p>Your two numbers sum up to: 3</p>
```

This is often a bad solution: it gets very difficult to manage with real services where the “state” of the server consists of much more than just a single number — 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). Worse, the state is all saved in the client browser — if it dies, then the interaction is gone. (Imagine doing your taxes, and praying that the browser won’t crash a third time.)

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.

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")))
```

and assuming that `web-read` works as a regular function, we need to begin with executing the first read:

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

We then need to take that result and plug it into an expression that will read the second number and sum the results — that’s the same as the first expression, except that instead of the first `web-read` we use a “hole”:

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

where `<*>` 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 the expression into a function:

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

We can split the second and third interactions in the same way. First we can assemble the above two bits of code into an expression that has the same meaning as the original one:

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

And now we can continue doing this and split the body of the consumer:

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

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

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

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

Doing all of this gives us:

```
((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:

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

except that the “real” thing would throw an error and die once the prompt is printed. The trick is one that we’ve already seen: we can turn the code inside-out by making the above “hole functions” be an argument to the reading function — a consumer callback for what needs to be done once the number is read. This callback is called a *continuation*, and we’ll use a `/k` suffix for names of functions that expect a continuation (`k` is a common name for a continuation argument):

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

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. It makes things a little easier, since we pass the hole function to `web-read/k`, and it will invoke it when needed:

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

You might notice that this looks too complicated; we could get exactly the same result with:

```
(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… So why have it? Remember that the main problem is that in the context of a web server we think of `web-read` as something that throws an error and kills the computation. So if we use such a `web-read/k` with a continuation, we can make it save this continuation in some global state, so it can be used later when there is a value.

As a side note, all of this might start looking very familiar to you if you have any experience working with callback-heavy code. In fact, consider the fact that the continuation (or `k`) is basically just a callback, so the above is roughly:

```
webRead("First number", function(a) {
  webRead("Second number", function(b) {
    webDisplay(a + b);
  });
});
```

We'll talk more about JavaScript later.

Simulating web reading

We can now actually try all of this in plain Racket by simulating web interactions. This is useful to look at the core problem while avoiding the whole web mess that is uninteresting for the purpose of our discussion. The main feature that we need to emulate is statelessness — and as we've discussed, we can simulate that using `error` to guarantee that the process is properly killed for each interaction. We will do this in `web-display` which simulates sending the results to the client and therefore terminates the server process:

```
(define (web-display n)
  (error 'web-display "~s" n))
```

More importantly, we need to do it in `web-read/k` — but in this case, we need more than just an `error` — we need a way to store the `k` so the computation can be resumed later. To continue with the web analogy we do this in two steps: `error` is used to display the information (the input prompt), and the user action of entering a number and submitting it will be simulated by calling a function. Since the computation is killed after we show the prompt, the way to implement this is by making the user call a toplevel `submit` function — and before throwing the interaction error, we'll save the `k` continuation in a global box:

```
(define (web-read/k prompt k)
  (set-box! resumer k)
  (error 'web-read
         "enter (submit N) to continue the following\n  ~a:"
         prompt))
```

`submit` uses the saved continuation:

```
(define (submit n)
  ((unbox resumer) n))
```

For safety, we'll initialize `resumer` with a function that throws an error (a real one, not intended for interactions), make `web-display` reset it to the same function, and also make `submit` do so after grabbing its value — meaning that `submit` can only be used after a `web-read/k`. And for convenience, we'll use `raise-user-error` instead of `error`, which is a Racket function that throws an error without a stack trace (since our errors are intended). It's also helpful to disable debugging in DrRacket, so it won't take us back to the code over and over again.

```
;; Fake web interaction library (to be used with manual code CPS-ing
;; examples)

#lang racket

(define error raise-user-error)

(define (nothing-to-do ignored)
  (error 'REAL-ERROR "No computation to resume."))

(define resumer (box nothing-to-do))

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

(define (web-read/k prompt k)
  (set-box! resumer k)
  (error 'web-read
         "enter (submit N) to continue the following\n  ~a:"
         prompt))

(define (submit n)
  ;; to avoid mistakes, we clear out `resumer' before invoking it
  (let ([k (unbox resumer)])
    (set-box! resumer nothing-to-do)
    (k n)))
```

We can now try out our code for the addition server, using plain argument names instead of `<*>`s:

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

and see how everything works. You can also try now the bogus expression that we mentioned:

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

and see how it breaks: the first `web-read/k` saves the identity function as the global resumer, losing the rest of the computation.

Again, this should be familiar: we've taken a simple compound expression and "linearized" it as a sequence of an input operation and a continuation receiver for its result. This is essentially the same thing that we used for dealing with inputs in the lazy language — and the similarity is not a coincidence. The problem that we faced there was very different (representing IO as values that describe it), but it originates from a similar situation — some computation goes on (in whatever way the lazy language decides to evaluate it), and when we have a need to read something we must return a description of this read that contains "the rest of the computation" to the eager part of the interpreter that executes the IO. Once we get the user input, we send it to this computation remainder, which can return another read request, and so on.

Based on this intuition, we can guess that this can work for any piece of code, and that we can even come up with a nicer "flat" syntax for it. For example, here is a simple macro that flattens a sequence of reads and a final display:

```
(define-syntax web-code
  (syntax-rules (read display)
    [(_ (read n prompt) more ...)
```

```
        (web-read/k prompt
          (lambda (n)
            (web-code more ...)))]
       [(_ (display last))
        (web-display last)]))
```

and using it:

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

However, we'll avoid such cuteness to make the transformation more explicit for the sake of the discussion. Eventually, we'll see how things can become even better than that (as done in Racket): we can get to write plain-looking Racket expressions and avoid even the need for an imperative form for the code. In fact, 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"))))
```

There is not much more than that — it has two utilities, `page` creates a well-formed web page, and `web-read` performs the reading. 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. Here's the full code:

```
#lang web-server/insta
(define (page . body)
  (response/xexpr
   `(html (body ,@(map (lambda (x)
                         (if (number? x) (format "~a" x) x))
                       body)))))
(define (web-read prompt)
  ((compose string->number (curry extract-binding/single 'n)
            request-bindings send/suspend)
   (lambda (k)
     (page `(form ([action ,k])
                  ,prompt ": " (input ([type "text"] [name "n"])))))))
(define (start initial-request)
  (page "The sum is: "
        (+ (web-read "First number")
           (web-read "Second number"))))
```

More Web Transformations

> 👉 ***PLAI §17***

Transforming a recursive function

Going back to transforming code, we did the transformation on a simple expression — and as you'd guess, it's possible to make it work for more complex code, even recursive functions. Let's start with some simple function that sums up a bunch of numbers, given a list of prompts for these numbers. Since it's a function, it's a reusable piece of code that can be used in multiple places, and to demonstrate that, we add a `web-display` with a specific list of prompts.

```
(define (sum prompts)
  (if (null? prompts)
    0
    (+ (web-read (first prompts))
       (sum (rest prompts)))))

(web-display (sum '("First" "Second" "Third")))
```

We begin by converting the `web-read` to its continuation version:

```
(define (sum prompts)
  (if (null? prompts)
    0
    (web-read/k (first prompts)
      (lambda (n)
        (+ n
           (sum (rest prompts)))))))

(web-display (sum '("First" "Second" "Third")))
```

But using `web-read/k` immediately terminates the running computation, which means that when `sum` is called on the last line, the surrounding `web-display` will be lost, and therefore this will not work. The way to solve this is to make `sum` itself take a continuation, which we'll get in a similar way — by rewriting it as a `sum/k` function, and then we can make our sample use pull in the web-display into the callback as we've done before:

```
(define (sum/k prompts k)
  (if (null? prompts)
    0
    (web-read/k (first prompts)
      (lambda (n)
        (+ n
           (sum (rest prompts)))))))

(sum/k '("First" "Second" "Third")
       (lambda (sum) (web-display sum)))
```

We also need to deal with the recursive `sum` call and change it to a `sum/k`. Clearly, the continuation is the same continuation that the original sum was called with, so we need to pass it on in the recursive call too:

```
(define (sum/k prompts k)
  (if (null? prompts)
    0
    (web-read/k (first prompts)
      ;; get the value provided by the user, and add it to the value
      ;; that the recursive call generates
      (lambda (n)
        (+ n
           (sum/k (rest prompts)
                  k))))))

(sum/k '("First" "Second" "Third")
       (lambda (sum) (web-display sum)))
```

But there is another problem now: the addition is done outside of the continuation, therefore it will be lost as soon as there's a second `web-read/k` call. In other words, computation bits that are outside of any continuations are going to disappear, and therefore they must be encoded as an explicit part of the continuation. The solution is therefore to move the addition *into* the continuation:

```
(define (sum/k prompts k)
  (if (null? prompts)
    0
    (web-read/k (first prompts)
      (lambda (n)
        (sum/k (rest prompts)
               (lambda (sum-of-rest)
                 (k (+ n sum-of-rest))))))))

(sum/k '("First" "Second" "Third")
       (lambda (sum) (web-display sum)))
```

Note that with this code every new continuation is bigger — it contains the previous continuation (note that “contains” here is done by making it part of the closure), and it also contains one new addition.

But if the continuation is only getting bigger, then how do we ever get a result out of this? Put differently, when we reach the end of the prompt list, what do we do? — Clearly, we just return 0, but that silently drops the continuation that we worked so hard to accumulate. This means that just returning 0 is wrong — instead, we should send the 0 to the pending continuation:

```
(define (sum/k prompts k)
  (if (null? prompts)
    (k 0)
    (web-read/k (first prompts)
      (lambda (n)
        (sum/k (rest prompts)
               (lambda (sum-of-rest)
                 (k (+ n sum-of-rest))))))))

(sum/k '("First" "Second" "Third")
       (lambda (sum) (web-display sum)))
```

This makes sense now, and the code works as expected. This `sum/k` is a utility to be used in a web server application, and such applications need to be transformed in a similar way to what we’re doing. Therefore, our own `sum/k` is a function that expects to be invoked from such transformed code — so it needs to have an argument for the waiting receiver, and it needs to pass that receiver around (accumulating more functionality into it) until it’s done.

As a side note, `web-display` is the only thing that is used in the toplevel continuation, so we could have used it directly without a `lambda` wrapper:

```
(sum/k '("First" "Second" "Third")
       web-display)
```

Using `sum/k`

To get some more experience with this transformation, we’ll try to convert some code that uses the above `sum/k`. For example, lets add a multiplier argument that will get multiplied by the sum of the given numbers. Begin with the simple code. This is an actual application, so we’re writing just an expression to do the computation and show the result, not a function.

```
(web-display (* (web-read "Multiplier")
                (sum '("First" "Second" "Third"))))
```

We now need to turn the two function calls into their `*/k` form. Since we covered `sum/k` just now, begin with that. The first step is to inspect its continuation: this is the same code after we replace the `sum` call with a hole:

```
(web-display (* (web-read "Multiplier")
                <*>))
```

Now take this expression, make it into a function by abstracting over the hole and call it `n`, and pass that to `sum/k`:

```
(sum/k '("First" "Second" "Third")
       (lambda (n)
         (web-display (* (web-read "Multiplier")
                         n))))
```

(Note that this is getting rather mechanical now.) Now for the `web-read` part, we need to identify its continuation — that's the expression that surrounds it inside the first continuation function, and we'll use `m` for the new hole:

```
         (* m
            n)
```

As above, abstract over `m` to get a continuation, and pass it into `web-read/k`:

```
(sum/k '("First" "Second" "Third")
       (lambda (n)
         (web-read/k "Multiplier"
                     (lambda (m)
                       (web-display (* m n))))))
```

and we're done. An interesting question here is what would happen if instead of the above, we start with the `web-read` and *then* get to the `sum`? We'd end up with a different version:

```
(web-read/k "Multiplier"
            (lambda (m)
              (sum/k '("First" "Second" "Third")
                     (lambda (n)
                       (web-display (* m n))))))
```

Note how these options differ — one reads the multiplier first, and the other reads it last.

> *Side-note: if in the last step of turning `web-read` to `web-read/k` we consider the whole expression when we formulate the continuation, then we get to the same code. But this isn't really right, since it is converting code that is already-converted.*

In other words, our conversion results in code that fixes a specific evaluation order for the original expression. The way that the inputs happen in the original expression

```
(web-display (* (web-read "Multiplier")
                (sum '("First" "Second" "Third"))))
```

is unspecified in the code — it only happens to be left-to-right implicitly, because Racket evaluates function arguments in that order. However, the converted code does *not* depend on how Racket evaluates function arguments. (Can you see a similar conclusion here about strictness?)

Note also another property of the converted code: every intermediate result has a name now. This makes sense, since another way to fix the evaluation order is to do just that. For example, convert the above to either

```
(let* ([m (web-read "Multiplier")]
       [n (sum '("First" "Second" "Third"))])
  (* m n))
```

or

```
(let* ([n (sum '("First" "Second" "Third"))]
       [m (web-read "Multiplier")])
  (* m n))
```

This is also a good way to see why this kind of conversion can be a useful tool in compiling code: the resulting code is in a kind of a low-level form that makes it easy to translate to assembly form, where function calls are eliminated, and instead there are only jumps (since all calls are tail-calls). In other words, the above can be seen as a piece of code that is close to something like:

```
val n = sum(["First","Second","Third"])
val m = web_read("Multiplier")
web_display(m*n)
```

and it's almost visible in the original converted code if we format it in a very weird way:

```
;; sum(["First","Second","Third"]) -> n
(sum/k '("First" "Second" "Third") (lambda (n)
;; web_read("Multiplier") -> m
(web-read/k "Multiplier" (lambda (m)
;; web_display(m*n)
(web-display (* m n))))))
```

Converting stateful code

Another case to consider is applying this transformation to code that uses mutation with some state. For example, here's some simple account code that keeps track of a `balance` state:

```
(define account
  (let ([balance (box 0)])
    (lambda ()
      (set-box! balance
                (+ (unbox balance)
                   (web-read (format "Balance: ~s; Change"
                                     (unbox balance)))))
      (account))))
```

(Note that there is no `web-display` here, since it's an infinite loop.) As usual, the fact that this function is expected to be used by a web application means that it should receive a continuation:

```
(define account/k
  (let ([balance (box 0)])
    (lambda (k)
      (set-box! balance
                (+ (unbox balance)
                   (web-read (format "Balance: ~s; Change"
                                     (unbox balance)))))
      (account))))
```

Again, we need to convert the `web-read` into `web-read/k` by abstracting out its continuation. We'll take the `set-box!` expression and create a continuation out of it:

```
      (set-box! balance
                (+ (unbox balance)
                   <*>))
```

and using `change` as the name for the continuation argument, we get:

```
(define account/k
  (let ([balance (box 0)])
```

```
    (lambda (k)
      (web-read/k (format "Balance: ~s; Change"
                          (unbox balance))
                  (lambda (change)
                    (set-box! balance (+ (unbox balance) change))))
      (account))))
```

And finally, we translate the loop call to pass along the same continuation it received (it seems suspicious, but there's nothing else that could be used there):

```
(define account/k
  (let ([balance (box 0)])
    (lambda (k)
      (web-read/k (format "Balance: ~s; Change" (unbox balance))
                  (lambda (change)
                    (set-box! balance (+ (unbox balance) change))))
      (account/k k))))
```

But if we try to run this — (account/k web-display) — we don't get any result at all: it reads one number and then just stops without the usual request to continue, and without showing any result. The lack of printed result is a hint for the problem — it must be the void return value of the `set-box!`. Again, we need to remember that invoking a `web-read/k` kills any pending computation and the following (resume) will restart its continuation — but the recursive call is not part of the loop.

The problem is the continuation that we formulated:

```
      (set-box! balance
                (+ (unbox balance)
                   change))
```

which should actually contain the recursive call too:

```
      (set-box! balance
                (+ (unbox balance)
                   change))
      (account/k k)
```

In other words, the recursive call was left outside of the continuation, and therefore it was lost when the fake server terminated the computation on a `web-read/k` — so it must move into the continuation as well:

```
(define account/k
  (let ([balance (box 0)])
    (lambda (k)
      (web-read/k (format "Balance: ~s; Change" (unbox balance))
                  (lambda (change)
                    (set-box! balance (+ (unbox balance) change))
                    (account/k k))))))
```

and the code now works. The only suspicious thing that we're still left with is the loop that passes `k` unchanged — but this actually is the right thing to do here. The original loop had a tail-recursive call that didn't pass along any new argument values, since the infinite loop is doing its job via mutations to the box and nothing else was done in the looping call. The continuation of the original call is therefore also the continuation of the second call, etc. All of these continuations are closing over a single box and this binding does not change (it *cannot* change if we don't use a `set!`); instead, the boxed value is what changes through the loop.

Converting higher order functions

Next we try an even more challenging transformation: a higher order function. To get a chance to see more interesting examples, we'll have some more code in this case.

For example, say that we want to compute the sum of squares of a list. First, the simple code (as above, there's no need to wrap a `web-display` around the whole thing, just make it return the result):

```
(define (sum l) (foldl + 0 l))
(define (square n) (* n n))
(define (read-number prompt)
  (web-read (format "~a number" prompt)))
(web-display (sum (map (lambda (prompt)
                         (square (read-number prompt)))
                       '("First" "Second" "Third"))))
```

Again, we can begin with `web-read` — we want to convert it to the continuation version, which means that we need to convert `read-number` to get one too. This transformation is refreshingly trivial:

```
(define (read-number/k prompt k)
  (web-read/k (format "~a number" prompt) k))
```

This is an interesting point — it's a simple definition that just passes `k` on, as is. The reason for this is similar to the simple continuation passing of the imperative loop: the pre-translation `read-number` is doing a simple tail call to `web-read`, so the evaluation context of the two is identical. The only difference is the prompt argument, and that's the same `format` call.

Of course things would be different if `format` itself required a web interaction, since then we'd need some `format/k`, but without that things are really simple. The same goes for the two utility functions — `sum` and `square`: they're not performing any web interaction so it seems likely that they'll stay the same.

We now get to the main expression, which should obviously change since it needs to call `read-number/k`, so it needs to send it some continuation. By now, it should be clear that passing an identity function as a continuation is going to break the surrounding context once the running computation is killed for the web interaction. We need to somehow generate a top-level identity continuation and propagate it inside, and the `sum` call should be in that continuation together with the `web-display` call. Actually, if we do the usual thing and write the expression with a `<*>` hole, we get:

```
(web-display (sum (map (lambda (prompt) (square <*>))
                       '("First" "Second" "Third"))))
```

and continuing with the mechanical transformation that we know, we need to abstract over this expression+hole into a function, then pass it as an argument to `read-number/k`:

```
;; very broken
(read-number/k
 (lambda (<*>)
   (web-display (sum (map (lambda (prompt) (square <*>))
                          '("First" "Second" "Third"))))))
```

But that can't work in this case — we need to send `read-number/k` a prompt, but we can't get a specific one since there is a *list* of them. In fact, this is related to a more serious problem — pulling out `read-number/k` like this is obviously broken since it means that it gets called only once, instead, we need to call it once for each prompt value.

The solution in this case is to convert `map` too:

```
(web-display (sum (map/k (lambda (prompt)
                           (square (read-number prompt)))
                         '("First" "Second" "Third")
                         ...some-continuation...)))
```

and of course we should move `web-display` and `sum` into that continuation:

```
(map/k (lambda (prompt) (square (read-number prompt)))
       '("First" "Second" "Third")
       (lambda (l) (web-display (sum l))))
```

We can now use `read-number/k`, but the question is what should it get for it's continuation?

```
(map/k (lambda (prompt) (square (read-number/k prompt ???)))
       '("First" "Second" "Third")
       (lambda (l) (web-display (sum l))))
```

Clearly, `map/k` will need to somehow communicate *some* continuation to the mapped function, which in turn will send it to `read-number/k`. This means that the mapped function should get converted too, and gain a `k` argument. To do this, we'll first make things convenient and have a name for it (this is only for convenience, we could just as well convert the `lambda` directly):

```
(define (read-squared prompt)
  (square (read-number/k prompt ???)))
(map/k read-squared
       '("First" "Second" "Third")
       (lambda (l) (web-display (sum l))))
```

Then convert it in the now-obvious way:

```
(define (read-squared/k prompt k)
  (read-number/k prompt
                 (lambda (n)
                   (k (square n)))))
(map/k read-squared/k
       '("First" "Second" "Third")
       (lambda (l) (web-display (sum l))))
```

Everything is in place now — except for `map/k`, of course. We'll start with the definition of plain `map`:

```
(define (map f l)
  (if (null? l)
    null
    (cons (f (first l)) (map f (rest l)))))
```

The first thing in turning it into a `map/k` is adding a `k` input,

```
(define (map f l k)
  (if (null? l)
    null
    (cons (f (first l)) (map f (rest l)))))
```

and now we need to face the fact that the `f` input is itself one with a continuation — an `f/k`:

```
(define (map/k f/k l k)
  (if (null? l)
    null
    (cons (f (first l)) (map f (rest l)))))
```

Consider now the single `f` call — that should turn into a call to `f/k` with some continuation:

```
(define (map/k f/k l k)
  (if (null? l)
    null
    (cons (f/k (first l) ???) (map f (rest l)))))
```

but since `f/k` will involve a web interaction, it will lead to killing the `cons` around it. The solution is to move that `cons` into the continuation that is handed to `f/k` — and as usual, this involves the second `cons` argument — the continuation is derived from replacing the `f/k` call by a hole:

```
    (cons <*> (map f (rest l)))
```

and abstracting that hole, we get:

```
(define (map/k f/k l k)
  (if (null? l)
    null
    (f/k (first l)
         (lambda (result)
           (cons result (map f (rest l)))))))
```

We now do exactly the same for the recursive `map` call — it should use `map/k` with `f/k` and some continuation:

```
(define (map/k f/k l k)
  (if (null? l)
    null
    (f/k (first l)
         (lambda (result)
           (cons result (map/k f/k (rest l) ???))))))
```

and we need to move the surrounding `cons` yet again into this continuation. The holed expression is:

```
           (cons result <*>)
```

and abstracting that and moving it into the `map/k` continuation we get:

```
(define (map/k f/k l k)
  (if (null? l)
    null
    (f/k (first l)
         (lambda (result)
           (map/k f/k (rest l)
                  (lambda (new-rest)
                    (cons result new-rest)))))))
```

There are just one more problem with this — the `k` argument is never used. This implies two changes, since it needs to be used once in each of the conditional branches. Can you see where it should be added? (Try to do this before reading the code below.)

The complete code follows:

```
(define (map/k f/k l k)
  (if (null? l)
    (k null)
    (f/k (first l)
         (lambda (result)
           (map/k f/k (rest l)
                  (lambda (new-rest)
                    (k (cons result new-rest)))))))))
(define (sum l) (foldl + 0 l))
(define (square n) (* n n))
(define (read-number/k prompt k)
  (web-read/k (format "~a number" prompt) k))
(define (read-squared/k prompt k)
  (read-number/k prompt (lambda (n) (k (square n)))))
(map/k read-squared/k
       '("First" "Second" "Third")
       (lambda (l) (web-display (sum l))))
```