Chapter 17

More Web Transformation

We have already seen how the application

\[
(\text{web-display}
  \left( + \left( \text{web-read} \ "\text{First number: } \" \right)
  \left( \text{web-read} \ "\text{Second number: } \" \right) \right)
\]

must be transformed into

\[
(\text{web-read/k} \ "\text{First number: } \"
  \left( \text{lambda} \ (\bullet_1)
    \left( \text{web-read/k} \ "\text{Second number: } \"
      \left( \text{lambda} \ (\bullet_2)
        \left( \text{web-display}
          \left( + \bullet_1 \bullet_2 \right) \right) \right) \right) \right)
\]

to execute on the Web. Let us now examine some more applications of a more complex flavor.

17.1 Transforming Recursive Code

Suppose we have the function \textit{tally}. It consumes a list of items and prompts the user for the cost of each item. When done, it generates the sum of these items. A programmer could, for instance, invoke this function to compute the sum, then display the result on the Web. The code for \textit{tally} is as follows:

\[
(\text{define} \ (\text{tally} \ \text{item-list})
  \left( \text{if} \ (\text{empty?} \ \text{item-list})
    \text{empty}
    \left( + \left( \text{web-read} \ (\text{generate-item-cost-prompt} \ (\text{first} \ \text{item-list})))
      \text{tally} \ (\text{rest} \ \text{item-list}))) \right) \right)
\]

This version of \textit{tally} is clearly not Web-friendly, due to the use of \textit{web-read}, which we do not know how to implement. We must therefore transform this code.

The first thing to observe is that on its own, \textit{tally} is not a complete program: it doesn’t do anything! Instead, it is a library function that may be used in many different contexts. Because it has a Web interaction,
however, there is the danger that at the point of interaction, the rest of the computation—i.e., the computation that invoked tally—will be lost. To prevent this, tally must consume an extra argument, a receiver, that represents the rest of the pending computation. To signify this change in contract, we will use the convention of appending /k to the name of the function and k to name the receiver parameter.

\[
\text{(define } (tally/k \text{ item-list } k) \\
\text{ (if } (\text{empty? item-list}) \\
\text{ empty } \\
(\text{+ } (\text{web-read (generate-item-cost-prompt (first item-list))}) \\
\text{ (tally (rest item-list)))))
\]

What is the first thing this function does? It checks for whether the list is empty. Does this involve any Web interaction? Clearly not; all the data and primitives are available locally. If the list is not empty, then tally prompts for a user input through the Web. This must happen through web-read/k. What is the receiver of this Web invocation? That is, what computation remains to be done? Clearly the recursive invocation of tally; but there is also the receiver, k, which represents the rest of the waiting computation. Therefore, the Web-friendly version of tally appears to become

\[
\text{(define } (tally/k \text{ item-list } k) \\
\text{ (if } (\text{empty? item-list}) \\
\text{ empty } \\
(\text{web-read/k (generate-item-cost-prompt (first item-list))} \\
\text{(lambda } (v) \\
\text{ (+ v} \\
\text{ (tally/k (rest item-list) k))))))))
\]

We can read the second argument to web-read/k as saying: “Consume the value provided by the user and add it to the value generated by the recursion. The receiver in the recursive is the same k as before, because the computation pending outside the function has not changed.”

This may look reasonable, but it suffers from an immediate problem. When the recursive call occurs, if the list had two or more elements, then there will immediately be another Web interaction. Because this will terminate the program, the pending addition will be lost! Therefore, the addition of v has to move into the receiver fed to tally/k! That is,

\[
\text{(define } (tally/k \text{ item-list } k) \\
\text{ (if } (\text{empty? item-list}) \\
\text{ empty } \\
(\text{web-read/k (generate-item-cost-prompt (first item-list))} \\
\text{(lambda } (\text{first-item-cost}) \\
\text{ (tally/k (rest item-list)} \\
\text{ (lambda } (\text{tally-of-remaining-costs}) \\
\text{ (k (+ first-item-cost} \\
\text{ tally-of-remaining-costs))))))))
\]
That is, the receiver of the Web interaction is invoked with the cost of the first item. When \( \text{tally}/k \) is invoked recursively, it is applied to the rest of the list. Its receiver must therefore receive the tally of costs of the remaining items. That explains the pattern in the receiver.

The only problem is, where does a receiver ever get a value? We create larger-and-larger receivers on each recursive invocation, but the only place we ever feed a value to a receiver is inside a procedure—how does that procedure get invoked in the first place?

Here is the same problem, but approached from an entirely different angle (that also answers the question above). Notice that each recursive invocation of \( \text{tally}/k \) takes place in the aftermath of a Web interaction. We have already seen how the act of Web interaction terminates the pending computation. Therefore, when the list empties, where is the value \( \text{empty} \) going? Presumably to the pending computation—but all that computation has now been recorded in \( k \), which is expecting a value. Therefore, the correct transformation of this function is

\[
(\text{define} (\text{tally}/k \text{ item-list } k))
\]
\[
(\text{if} (\text{empty? item-list})
\]
\[
(\text{empty}/k)
\]
\[
(\text{web-read}/k (\text{generate-item-cost-prompt} (\text{first item-list}))
\]
\[
(\text{lambda} (\text{first-item-cost})
\]
\[
(\text{tally}/k (\text{rest item-list})
\]
\[
(\text{lambda} (\text{tally-of-remaining-costs})
\]
\[
(k (+ \text{first-item-cost}
\]
\[
\text{tally-of-remaining-costs}))))
\]

Now we have a truly reusable abstraction. Whatever the computation pending outside the invocation of \( \text{tally}/k \), its proper Web transformation yields a receiver. If this receiver is fed as the second parameter to \( \text{tally}/k \), then it is guaranteed to be invoked with the value that \( \text{tally} \) would have produced in a non-Web (e.g., console) interaction. The pattern of receiver creation within \( \text{tally}/k \) ensures that no pending computation gets lost due to the behavior of the Web protocol.

**Exercise 17.1.1** There is a strong formal claim hidden behind this manual transformation: that the value given to the initial \( k \) fed to \( \text{tally}/k \) is the same as that returned by \( \text{tally} \) in the non-Web version. Prove this.

### 17.2 Transforming Multiple Functions

Suppose we have the functions \( \text{total}+s&h \). It consumes a list of items to purchase, queries the user for the cost of each item, then generates another prompt for the corresponding shipping-and-handling cost\(^1\) and finally prints the result of adding these together. The function \( \text{total}+s&h \) relies on \( \text{tally} \) to compute the sum of the goods alone.

\[
(\text{define} (\text{total}+s&h \text{ item-list})
\]
\[
(\text{local} [(\text{define} \text{total} (\text{tally item-list}))]
\]

\(^1\)The term *shipping and handling* refers to a cost levied in the USA by companies that handle long-distance product orders placed by the mail, phone and Internet. It is ostensibly the price of materials to package and labor to dispatch the ordered goods. This rate is usually a (step) function of the cost of items ordered, and must hence be calculated at the end of the transaction.
Just as we argued in the transformation of \textit{tally}, this function alone does not constitute a computation. It must therefore consume an extra parameter, representing a receiver that will consume its result. Likewise, it cannot invoke \textit{tally}, because the latter performs a Web interaction; it must instead invoke \textit{tally/k}, passing along a suitable receiver to ensure no computation is lost.

\begin{verbatim}
(define (total+s&h/k item-list k)
  (local ([define total (tally/k item-list ??)])
    (+ (web-read (generate-s&h-prompt total))
      total)))
\end{verbatim}

Reasoning as before, what is the first thing \textit{total+s&h/k} does? It invokes a function to compute the tally. Because this function involves a Web interaction, it must be invoked appropriately. That is, the transformed function must take the form

\begin{verbatim}
(define (total+s&h/k item-list k)
  (tally/k item-list
    (lambda (tally-of-items)
      ??)))
\end{verbatim}

What is the pending computation? It is to bind the resulting value to \textit{total}, then perform another Web interaction:

\begin{verbatim}
(define (total+s&h/k item-list k)
  (tally/k item-list
    (lambda (tally-of-items)
      (local ([define total tally-of-items])
        ??)))))
\end{verbatim}

(Notice that the Web transformation has forced us to give names to intermediate results, thereby rendering the name \textit{total} unnecessary. We will, however, leave it in the transformed program so that the transformation appears as mechanical as possible.) With the pending computation, this is

\begin{verbatim}
(define (total+s&h/k item-list k)
  (tally/k item-list
    (lambda (tally-of-items)
      (local ([define total tally-of-items])
        (web-read/k (generate-s&h-prompt total)
          (lambda (s&h-amount)
            (k (+ s&h-amount
              total)))))))
\end{verbatim}

Notice how \textit{total+s&h/k} had to create a receiver to pass to \textit{tally/k}, the transformed version of \textit{tally}. Reading this receiver, it says to consume the value computed by \textit{tally/k} (in \textit{tally-of-items}), bind it, ask the user to enter the shipping-and-handling amount, compute the final total, and convey this amount to the initial receiver.
17.3. TRANSFORMING STATE

It’s easy to forget this last step: to apply $k$, the initial receiver supplied to $total + s & h / k$, to the final value. Doing so would effectively “forget” all the computation that was waiting for the result of $total + s & h / k$, i.e., the computation awaiting the result of $total + s & h$ in the original program. This is obviously undesirable.

You might worry that the local might be “forgotten” by the web-read/$k$ that follows. But all we care about is that the name total be associated with its value, and the receiver will take care of that (since it is a closure, it must be closed over the value of total).

17.3 Transforming State

Suppose we want to write a program that keeps track of an account’s balance. On every invocation it presents the current balance and asks the user for a change (i.e., deposit or withdrawal, represented respectively by positive and negative numbers). In principle, the Web application might look like this:

```
(define account
  (local ([define balance 0])
    (lambda ()
      (begin
        (set! balance (+ balance
                      (web-read
                        (format "Balance: ~a; Change" balance))))
      (account))))))
```

Note that account is bound to a closure, which holds a reference to balance. Recall that mutable variables introduce a distinction between their location and the value at that location. The closure closes over the location, while the store is free to mutate underneath. Thus, even though balance always refers to the same location, its value (the actual account balance) changes with each interaction.

How do we transform this program? Clearly the procedure bound to account must take an additional argument to represent the remainder of the computation:

```
(define account/k
  (local ([define balance 0])
    (lambda (k)
      (begin
        (set! balance (+ balance
                      (web-read
                        (format "Balance: ~a; Change" balance))))
      (account/k ???))))))
```

More importantly, we must move the web-read to be the first action in the procedure:

```
(define account/k
  (local ([define balance 0])
    (lambda (k)
      (begin
        (web-read/k (format "Balance: ~a; Change" balance)
```
(lambda (v)
  (begin
    (set! balance (+ balance v))
    (account/k))))))

What’s left is to determine what argument to pass as the receiver in the recursive call. What new pending activity have we created? The only thing the function does on each recursion is to mutate balance, which is already being done in the receiver to the Web interaction primitive. Therefore, the only pending work is whatever was waiting to be done before invoking account/k. This results in the following code:

(define account/k
  (local ([define balance 0])
    (lambda (k)
      (begin
        (web-read/k (format "Temperature in city \"a\" c") k)
        (begin
          (set! balance (+ balance v))
          (account/k k))))))

The closure created as the receiver for the Web interaction has a key property: it closes over the location of balance, not the value. The value itself is stored in the heap memory that is kept alive by the Web server.

Exercise 17.3.1 If we wanted to run this application without any reliance on a custom server (Section 16.4), we would have to put these heap data somewhere else. Can we put them in hidden fields, as we discussed in Section 16?

17.4 Transforming Higher-Order Functions

Suppose our Web program were the following:

(define (get-one-temp c)
  (web-read (format "Temperature in city \"a\" c")))

(web-display
  (average
    (map get-one-temp
      (list "Bangalore" "Budapest" "Houston" "Providence"))))

(Assume we’ve define average elsewhere.) In principle, converting this program is merely an application of what we studied in Section 17.1 and Section 17.2 but we’ll work through the details to reinforced what you read earlier.

Transforming get-one-temp is straightforward:

(define (get-one-temp/k c k)
  (web-read/k (format "Temperature in city \"a\" c"
    k)))
This means we must invoke this modified procedure in the `map`. We might thus try

```scheme
(web-display
 (average
  (map get-one-temp/k
   (list "Bangalore" "Budapest" "Houston" "Providence"))))
```

Unfortunately, `map` is expecting its first argument, the procedure, to consume only the elements of the list; it does not provide the second argument that `get-one-temp/k` needs. So Scheme reports

`map: arity mismatch for procedure get-one-temp/k: expects 2 arguments, given 1`

It therefore becomes clear that we must modify `map` also. Let’s first write `map` in full:

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

Clearly we must somehow modify the invocation of `f`. What can we pass as a second argument? Here’s one attempt:

```scheme
(define (map f l)
  (if (empty? l)
      empty
      (cons (f (first l) (lambda (x) x))
            (map f (rest l)))))
```

That is, we’ll pass along the identity function. Does that work? Think about this for a moment.

Let’s try testing it. We get the following interaction:

```
Welcome to DrScheme, version 208p1.
Language: PLAI - Advanced Student.
web-read/k: run (resume) to enter number and simulate clicking Submit
>
This means the program has arrived at `web-read/k` for the first time. We run

```scheme
> (resume)
```

which prompts us for an input. Suppose we enter 25. We then see

```
Temperature in city Bangalore: 25
25
>
```
It stopped: the program terminated without ever giving us a second Web prompt and asking us for the temperature in another city!

Why? Because the value of the receiver stored in the hash table or box is the identity function. When computation resumes (on the user’s submission), we expect to find the closure representing the rest of the computation. Since the stored closure is instead just the identity function, the program terminates thinking its task is done.

This gives us a pretty strong hint: the receiver we pass had better make some reference to map, and indeed, had better continue the iteration. In fact, let’s think about where we get the first value for cons. This value is the temperature for a city. It must therefore come from web-read/k. But that is exactly the value that web-read/k supplies to its receiver. Therefore, everything starting with the cons onward must move to within the closure:

\[
\text{(define (map f/k l))}
\]

\[
\text{(if (empty? l)}
\]

\[
\text{empty}
\]

\[
\text{(f/k (first l)}
\]

\[
\text{(lambda (v)}
\]

\[
\text{(cons v}
\]

\[
\text{(map f (rest l))))))}
\]

This version is still not quite okay. This is because the recursive call invokes map, which suffers from the same problem we have just discussed above. Indeed, running this version terminates after reading the temperature for the second city, and returns just a list containing the second city’s temperature! Instead, it must invoke a modified version of map, namely map/k, with an appropriate additional argument:

\[
\text{(define (map/k f/k l k)}
\]

\[
\text{(if (empty? l)}
\]

\[
\text{empty}
\]

\[
\text{(f/k (first l)}
\]

\[
\text{(lambda (v)}
\]

\[
\text{(cons v}
\]

\[
\text{(map f (rest l))}}))
\]

We must determine what to pass as an argument in the recursive call. But before we do that, let’s study this program carefully. When the first Web interaction results in a response, the server will invoke the (lambda (v) · · ·). This conses the input temperature to the value of the recursive call. The recursive call will, however, eventually result in an invocation of web-read/k. That invocation will halt the program. Once the program halts, we lose record of the cons. So this program can’t work either! We must instead move the cons inside the receiver, where it won’t be “forgotten”.

Using the intuition that the value given to each receiver is the result of computing the function on its other arguments, it makes sense to think of the value given on invoking map/k on the rest of the list as the list of temperatures for the remaining cities. Therefore, we simply need to cons the temperature for the first city onto this result:

\[
\text{(define (map/k f/k l k)}
\]
17.4. TRANSFORMING HIGHER-ORDER FUNCTIONS

(if (empty? l)
  empty
  (f/k (first l)
    (lambda (v)
      (map/k f/k (rest l)
        (lambda (v-rest)
          (cons v v-rest)))))))

Now we're ready to modify the main program. We had previously written

(web-display
  (average
    (map get-one-temp/k
      (list "Bangalore" "Budapest" "Houston" "Providence"))))

We have to convert the invocation of map to one of map/k, and in turn, determine what to pass as the second argument to map/k. Using the same reasoning we have employed before (in particular, that as written, the web-display and average will never execute, since they will be forgotten when the server terminates the program), we know to write this:

(map/k get-one-temp/k
  (list "Bangalore" "Budapest" "Houston" "Providence")
  (lambda (v)
    (web-display
      (average v)))))

This program now runs through the four cities, accepts the temperatures in order, and produces . . . the empty list.

What went wrong here? We can reason this way. The empty list cannot result from average (which must produce a number), so we can reason that the initial receiver must never have been invoked at all. (We can verify this by commenting out the definition of average and noticing that this doesn’t cause a problem: the procedure is never invoked.) So it must be the case that the receiver supplied to map/k never made it any further.

Studying map/k, we see the problem. Though the procedure consumes a receiver, that receiver never gets used anywhere in its body. In fact, we should be passing the result of the cons to this procedure:

(define (map/k f/k l k)
  (if (empty? l)
    empty
    (f/k (first l)
      (lambda (v)
        (map/k f/k (rest l)
          (lambda (v-rest)
            (k (cons v v-rest))))))))

Everything now looks hunky-dory, so we run the program, enter the four temperatures, and still get . . . the empty list!
Since there is really only one place in the program where we explicitly mention the empty list, we might suspect it now. Indeed, the first branch in the conditional of \texttt{map/k} is indeed the culprit. When a value becomes available, we should not return it. Why not? Because we know no procedure is awaiting it directly. Why not? Because according to the Web protocol, any waiting procedures would have terminated when the whole program terminated at the previous interaction point! Therefore, to return a value, a procedure must instead \textit{hand the value to the receiver}. That is,

\begin{verbatim}
(define (map/k f/k l k)
  (if (empty? l)
      (k empty)
      (f/k (first l)
           (lambda (v)
                    (map/k f/k (rest l)
                              (lambda (v-rest)
                                      (k (cons v v-rest)))))))))
\end{verbatim}

The moral of this lengthy story is that, to make a program Web-ready, we must (a) generate receivers that capture pending computations, and (b) pass values to receivers instead of returning them. In rare cases, a procedure will neither return a value nor generate additional pending computation—\texttt{get-one-temp} is a good example—in which case, its transformed version will consume a receiver and pass along the same receiver to other computations (as \texttt{get-one-temp/k} does).

\textbf{Exercise 17.4.1} Why did we not transform average? In general, what principle guides whether or not we transform a given procedure? (Make sure your principle also applies to map!)

\section{17.5 Perspective on the Web Transformation}

Notice three implications of the transformation the Web forces us to employ:

1. We have had to make decisions about the order of evaluation. That is, we had to choose whether to evaluate the left or the right argument of addition first. This was an issue we had specified only implicitly earlier; if our evaluator had chosen to evaluate arguments right-to-left, the Web program at the beginning of this document would have asked for the second argument before the first! We have made this left-to-right order of evaluation explicit in our transformation.

2. The transformation we use is global, namely it (potentially) affects all the procedures in the program by forcing them all to consume an extra receiver as an argument. We usually don’t have a choice as to whether or not to transform a procedure. Suppose \texttt{f} invokes \texttt{g} and \texttt{g} invokes \texttt{h}, and we transform \texttt{f} to \texttt{f/k} but don’t transform \texttt{g} or \texttt{h}. Now when \texttt{f/k} invokes \texttt{g} and \texttt{g} invokes \texttt{h}, suppose \texttt{h} consumes input from the Web. At this point the program terminates, but the last receiver procedure (necessary to resume the computation when the user supplies an input) is the one given to \texttt{f/k}, with all record of \texttt{g} and \texttt{h} erased\footnote{Indeed, we would have encountered an error even earlier, when the transformed version of \texttt{f}, namely \texttt{f/k}, tried to invoke \texttt{g} with an extra receiver argument that \texttt{g} was not transformed to accept. In this even simpler way, therefore, the transformation process has a cascading effect.}
3. This transformation sequentializes the program. Given a nested expression, it forces the programmer to choose which sub-expression to evaluate first (a consequence of the first point above); further, every subsequent operation lies in the receiver, which in turn picks the first expression to evaluate, pushing all other operations into its receiver; and so forth. The net result is a program that looks an awful lot like a traditional procedural program. This suggests that this series of transformations can be used to compile a program in a language like Scheme into one in a language like C! We will return to this point in Section 19.

Exercise 17.5.1 This presentation has intentionally left out the contracts on the functions. Add contracts to all the functions—both the original programs and the Web versions.

Exercise 17.5.2 Adding contracts to the Web versions (Exercise 17.5.1) reveals a very interesting pattern in the types of the receivers. Do you see a connection between this pattern and the behavior of the Web?