Converting Programs for the Web
Shriram Krishnamurthi
2001-10-03

1 Introduction

In the previous section, we studied a preliminary process by which we can convert programs written in a direct style for use on the Web. In this lecture and the next, we will study the process of doing this more rigorously—not by relying on our intuitions of where we “may” or “should” split functions, for instance, but where we must, and how to do it.

2 A Simple Example

We’ll begin with a very simple motivating example (in Scheme):

\[
\begin{align*}
(output &= \text{"The answer is "} \\
&
\quad (+ (prompt-read \ "First number: ") \\
&\quad (prompt-read \ "Second number: "))) \\
\end{align*}
\]

where \(output\) is some function that generates output and \(prompt-read\) is a corresponding function that prompts for and accepts input. This program isn’t very exciting, but for that very reason it makes a good case study.

Let’s think about converting this to run on the Web. We’ll assume that every invocation of \(prompt-read\) turns into a Web form prompting for an input. We know by now that the CGI protocol forces programs to interact with large granularity: they must generate forms, and wait for the user to submit the form before they process input. In particular, therefore, a Web program must perform one (perhaps pretty high-level) action, then stop and wait for the user to provide input. At that point the program starts again to perform the next action, and so on.

What’s the first action in this program? This is a tricky question with lots of reasonable answers. It may seem that \(output\) is “first”, because it’s clearly the first thing the Scheme evaluator begins to run. Or is it? Remember that in Scheme, we reduce arguments to values, which means that before \(output\) can begin to run, we must obtain values for the two arguments. The first is a string, which is already a value. The second is an addition, which in turn has two arguments, and so forth. Therefore, we have to think about this a bit instead of merely proceeding in textual order.
Which argument to + evaluates first? Well, we haven’t discussed this aspect of Scheme. In fact, DrScheme evaluates arguments from left-to-right. That is, we do in fact read the first number before the second. That means, the first action to complete in this program is

(prompt-read "First number: ")

For Web conversion, we’re interested in complete actions at a time, because we can’t suspend actions mid-way (since the CGI protocol forces the program to halt). Therefore, we agree this is the first complete action. Likewise, the second action to complete is (prompt-read "Second number: ") followed by the addition, followed, at last, by the invocation of output.

Okay, we’ve established the order of operations. It’s clear from this that our first CGI program should do the equivalent of

(prompt-read "First number: ")

Then what? Obviously the CGI program has to halt there, but surely we can’t just let that be the end of it, or we’ll never get around to doing the addition! Instead, it makes sense to record what operations are left, so if the user chooses to submit a response, someone or something will know what to do next. (For now, we’ll ignore how or where to record this. We will return to this issue later, so just take it on faith now.)

How do we record this? Well, we can always use text:

• read a value with the prompt “Second number: ”
• add the two
• display the output with the banner …

but it should be clear this isn’t very practical. Another thing we could do is create some elaborate script:

read "Second number: " value_2
value_3 = value_1 + value_2
display "The answer is " value_3

Fine, but now we’re stuck writing an evaluator for this new language (and, for that matter, figuring out what it means in the first place—for instance, where did value_1 come from above?).

The script is going in the right direction, running aground on the problem of defining a new language. But who says we need to do that? We already have a perfectly good language for describing computations, so let’s just use the one we have! That is, we can use Scheme to describe the remaining computation.

What kind of Scheme value should we use? Obviously not a number (which one?) or symbol (ditto); we could build an elaborate structure, but it’s not immediately clear what to pick (roughly the same problem as defining a new scripting language). What do we want to represent in the first place? Roughly this:
The answer is 

(+ □

(prompt-read "Second number: "))

where □ is replaced with the value the user provides to the form. If the user provides 5, we have the new Scheme program

(output "The answer is 

(+ 5

(prompt-read "Second number: ")))

which we can again factor out into the next complete action, etc.

Okay, so back to the problem: what kind of value do we use? Well, it’s clear we want to accept a value and substitute it for □. But that’s exactly what a function already does! Therefore, we could just write

(lambda (□)

(output "The answer is 

(+ □

(prompt-read "Second number: ")))

to represent the rest of the computation. We invoke this procedure with whatever value the user submits. That is, we can take our original program and rewrite it as

(prompt-read/web

"First number: "

(lambda (□)

(output "The answer is 

(+ □

(prompt-read "Second number: "))))

The definition prompt-read/web is, of course, pure fantasy at this point. We assume that it generates a form for a single value, writes its second argument somewhere, and halts. When the user provides a value, prompt-read/web awakens, accepts the user’s inputs, extracts the user’s value from the form bindings, retrieves the procedure, and feeds the user’s value to the procedure. In short, it looks something like¹

(define (prompt-read/web prompt action)

(let ([bindings ;; Assume, magically, we halt here and later resume

(generate-web-form prompt)])

(action (extract-binding 'user-input bindings))))

Let’s set prompt-read/web aside too, for now. (Again, I promise we’ll return to the definition of this later. Do keep count of my promises!) What happens when this procedure invokes its second primitive? We can’t get far because we

¹We name the argument action because it really represents the behavior of the script named in the HTML form’s action field.
have to generate another Web form—which means we have to go through the
rewriting process above once again. The resulting program now looks like

(prompt-read/web
 "First number: 
(lambda (□₁)
 (prompt-read/web
  "Second number: 
  (lambda (□₂)
    (output "The answer is 
    (+ □₁ □₂))))))

Notice that by the time we get around to performing the additions, we have
only values left (because argument expressions are always reduced to values by
Scheme before evaluating the procedure’s body). Likewise, both arguments to
prompt-read/web are values, at both call sites. The only violation of this pat-
tern is output’s second argument, which is an expression (the addition) rather
than a value. We excuse this for now because + is a primitive Scheme opera-
tion that does not itself perform any Web operations (which require messy
suspend/resume operations), nor invoke any procedures that do. Since we’re
doing all this transformation to manage Web interaction, we ignore addition.
Later (much later), we’ll see another reason why we can excuse lowly Scheme
primitives.

3 More Examples

1. Let’s examine a more complex example. Suppose we have

(define (countdown n)
  (cond
    [(zero? n) (output "Liftoff!")]
    [else
      (begin
        (prompt-read (format-print "(- t "a) and counting ..." n))
        (countdown (n 1)))]))

(countdown (prompt-read "Time left on launch pad: "))

This program resides at the heart of a high-tech Web interface to a shuttle
launcher. The program asks the CAPCOM to indicate how much time remains
until liftoff. Roughly once every second (DrScheme is pretty slow . . . ) it in-
dicates how much time is left and asks the CAPCOM to submit a form in
acknowledgment. It throws away any input the CAPCOM may have provided,
and continues on its relentless countdown, culminating at liftoff.²

²You can define format-print in terms of fprintf and current-output-port. If you don’t see
how to do it, ask for help.
Let’s try to transform this. It’s pretty clear that the very first complete action is

(prompt-read "Time left on launch pad: ")

so the program immediately becomes

(prompt-read/web
 "Time left on launch pad: 
 (lambda (□)
  (countdown □)))

It’s obvious we need to transform countdown, but less obvious how, so let’s focus on that alone. We leave the conditional alone—a resumed Web program is certainly welcome to test a conditional before it generates a new page. Let’s examine each branch of the conditional in turn. (zero? n) is a simple application of a primitive to a value, so we can use our primitive exemption. Similarly, there’s nothing to do with else. By the same token, we can leave the first response, (output "Liftoff!"), alone. This focuses our attention on the response to the second condition. But by now, it’s pretty clear how to handle prompt-read:

(prompt-read/web
 (format-print "(- t "a) and counting ..." n)
 (lambda (hole)
  (begin
   hole
   (countdown (− n 1))))))

The second argument to prompt-read/web nicely illustrates what happens to the CAPCOM’s input: we don’t use the value for any meaningful purpose (as the first term in begin) and proceed to the recursion. Thus, we get

(define (countdown n)
  (cond
   [(zero? n) (output "Liftoff!")]
   [else (prompt-read/web
     (format-print "(- t "a) and counting ..." n)
     (lambda (hole)
      (begin
       hole
       (countdown (− n 1)))))]))

(prompt-read/web
 "Time left on launch pad: 
 (lambda (□)
  (countdown □)))
2. Given the delays Shuttle launches frequently encounter, however, a more useful, if prosaic, procedure might be one that counts how long the Shuttle has stayed on the pad. That is, say the shuttle launcher interface now incorporates

\[
\text{(define (count-delay)}
\begin{align*}
&\text{(let ([new-delay (prompt-read "Delay: ")])} \\
&\text{(cond} \\
&\phantom{\text{(let ([new-delay (prompt-read "Delay: ")])}}\text{[(zero? new-delay) 0]} \\
&\phantom{\text{(let ([new-delay (prompt-read "Delay: ")])}}\text{[else (+ new-delay (count-delay))])))}
\end{align*}
\]

\[
\text{(output "Total wait: " (count-delay))}
\]

This program begins to run when the Shuttle is first held on the launchpad. It consumes a delay duration, and keeps consuming these every time there is an additional delay. When the Shuttle finally launches (indicated by an input of 0), it prints the total wait the crew had to endure. This program is clearly a bit more complex than the previous one, and it’s worth thinking a little about how you might convert it before reading on below.

Okay, let’s start. We immediately run into a problem: the first complete action is the invocation of count-delay, which must finish before we can output anything. But count-delay is no simple Scheme primitive; to the contrary, it invokes the dreaded prompt-read numerous times, so we need to determine how to stop and resume it using prompt-read/web.

Let’s return to our principle of doing first things first. We’ve just said that count-delay must complete before output can begin to run. This should sound familiar: we said that of prompt-read, and used it to motivate the creation of prompt-read/web. Why don’t we apply the same technique again? That is, the main expression becomes

\[
\text{(count-delay/web (lambda (square) (output "Total wait: " (square))}}
\]

From count-delay, we shall derive the Web-friendly function count-delay/web. This will take one extra argument, a procedure, representing what needs to happen when it’s done. When count-delay/web is done, it’ll feed its answer to that extra argument.

Transforming count-delay is a little more tricky than transforming countdown. First of all, the body becomes

\[
\text{(define (count-delay/web action)}
\begin{align*}
&\text{(prompt-read/web "Delay: ")} \\
&\text{(lambda (square))}
\end{align*}
\]

\[3\text{Why didn’t we raise the same fuss for the use of countdown that we do for the use of count-delay? Because no procedure was awaiting the output of countdown, we glossed over this aspect of its behavior. Technically, we should have done all that follows.}\]
We can consider let, when it binds a name to something already reduced to a value (as must be), to be a primitive. (Again, we’ll justify why much, much later. Another promise!) We’ve discussed most of the cond transformation already. Let’s discuss the two responses in more detail.

In the original program, the response to a zero delay is 0. Previously, we returned this to whichever expression invoked count-delay. Now, however, we’re in count-delay/web, and there’s no expression really awaiting its response. Because any such waiting invocation would be lost when the CGI program shuts down after generating the form, all that waiting code has been packaged into a procedure, which now represents the action argument. Let’s follow the routine: just as prompt-read/web provided its user-supplied value to its second argument, so should count-delay/web be a well-behaved “/web” procedure. Therefore, the first response becomes

\[
\text{(action 0)}
\]

By the same argument, the second response becomes

\[
\text{(action (+ new-delay (count-delay)))}
\]

The first response clearly applies action to a value. The second is rather more complex. We can’t apply action before adding, and the addition can’t complete until count-delay does, which involves form interactions—which halt the program, meaning we’ll never return to the pending +. Since we don’t want to lose track of the addition, we’re better off using count-delay/web instead, keeping track of the addition for future use when the user actually provides a value. Therefore, we can fill the gap with

\[
\text{(define (count-delay/web action)}
\]

\[
\text{(prompt-read/web “Delay: ” (lambda (□₁)}
\]

\[
\text{(let ([new-delay □₁])}
\]

\[
\text{(cond}
\]

\[
\text{[(zero? new-delay) (action 0)]}
\]

\[
\text{[else (count-delay/web (lambda (□₂)}
\]

\[
\text{(action (+ new-delay □₂)))]))))}
\]

And with that, we’re done! We’ve transformed the entire program. All procedures (including primitives) apply to values through, at most, a small number of nested primitive invocations (which we can assume complete almost immediately and do nothing clever). Any time we need to generate a Web page, we append an extra argument to all procedures on the “critical path” to keep track
of pending work. When prompt-read/web (magically) resumes with a user’s input, it invokes its second argument with that value to resume the rest of the computation.

3. Let’s examine one last example of this sort. As you know, temperature calculations are critical for Shuttle safety. We’ll present one more excerpt from the top-secret Shuttle launch software. Here’s the initial program:

```
(for-each (lambda (temp)
    (output "Converted (F):"
      (c->f temp)))
  (prompt-read "Temps (list of C): "))
```

This reads a list of temperatures °C, and converts them to °F, presenting the output one-per-page.4

As always, we’ll begin with the main program. Clearly, we first read a bunch of temperatures. Thus:

```
(prompt-read/web
 "Temps (list of C): ")
(lambda (λ)
  (for-each (lambda (temp)
      (output "Converted (F):"
        (c->f temp)))
    (λ))
)
```

This forces us to confront something we’ve swept under the rug until now: that output is another of these insidious Web procedures. That is, producing output forces the computation to halt (until the user clicks on, say, a Continue button). In this respect, it’s very much like prompt-read. We ignored it until now because calling output was always the very last thing our programs did, so it didn’t matter much whether that invocation returned or not. Now that we’re calling it in the midst of a loop, however, it very much matters if the computation doesn’t resume, because this means we’ll never seen most of the results.

All this indicates we really should be using output/web instead, and passing it one more argument:

```
(prompt-read/web
 "Temps (list of C): ")
(lambda (λ₁)
  (for-each (lambda (temp)
      (output/web "Converted (F):"
        (c->f temp))
    (λ₁))
)
```

---

4 For each is like map, except we don’t care about the value the procedural argument returns. When it’s done with the list, for-each returns some junk value that we mustn’t rely on.
But what do we put in this second argument? It represents the computation that remains. But what is left? This looks pretty complicated. We’re really invoking the procedure deep in the midst of for-each. For reference, it might help to see the definition of for-each:

\[
\text{(define (for-each } f \ l) \\\n\text{ (cond \\\n\text{ [(empty? } l) \text{'irrelevant}] \\\n\text{ [(cons? } l) \\\n\text{ (begin \\\n\text{ (f (first } l)) \\\n\text{ (for-each } f \text{ (rest } l))]})])}
\]

That is, the \text{(lambda} (\text{temp} \ldots) \text{)} is being invoked on the first element of the list in each iteration. What’s left is the rest of the iteration. But how can the \text{(lambda} (\square_2 \ldots) \text{)} possibly know about the iteration status of the for-each?

The answer is, \text{it can’t}—at least, not unless for-each gives it this information. But for for-each to indicate what’s left of its iteration, it too needs to become a “/web” procedure. That is,\text{5}

\[
\text{(define (for-each/web} f \ l \ action) \\\n\text{ (cond \\\n\text{ [(empty? } l) \text{(action 'irrelevant)]} \\\n\text{ [(cons? } l) \\\n\text{ (begin \\\n\text{ (f (first } l)) \\\n\text{ (for-each/web } f \text{ (rest } l) \text{ action)]]})])}
\]

The last line is fine: once we’re done invoking \(f\) on the first element, what remains to be done is to invoke it on the rest of the list, after which we continue with whatever was waiting for the loop to complete. Therefore, it’s reasonable that \text{action} should remain unchanged through the iterations.

However, doing this hasn’t accomplished our goal, which is to provide a representation of the remaining work to output/web. As it stands, when output/web returns, it isn’t going to know about the iterations that remain. With a little inspection, we realize the problem is that the functional argument to the loop pretends that it isn’t in a Web context, but it clearly is (indeed, it’s the one invoking the Web primitive). Therefore, the “Webification” of for-each induces a similar transformation on its first argument also.\text{6}

\[
\text{(define (for-each/web } f/web \ l \ action) \\\n\text{ (cond \\\n\text{ [(empty? } l) \text{(action 'irrelevant)]}}
\]

\text{Quick: do this as an exercise!}

\text{Notice our use of the /web convention to indicate the change in the contract of a Web-converted procedure.}
Now, finally, we can fill in the main expression:

\[ \text{prompt-read/web} \]

\[ \text{"Temps (list of C): "} \]

\[ \text{(lambda (□₁)} \]

\[ \text{(for-each/web (lambda (temp action))} \]

\[ \text{(output/web "Converted (F): "} \]

\[ \text{(c→f temp)} \]

\[ \text{(lambda (□₂)} \]

\[ \text{(action □₂))})} \]

\[ □₁ \]

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

That is, whatever (junk) value output/web supplies for □₂ flows to the action argument—which is of the form

\[ \text{(lambda (□)} \]

\[ \text{(begin} \]

\[ □ \]

\[ \text{(for-each/web f/web (rest l) action))})} \]

This throws away the junk value, and continues to iterate on the rest of the list, just as we would want. Note that the action for for-each/web is \( \text{(lambda (x) x)} \), reflecting that in the original, when the for-each completes, there’s nothing left to do. Some people might prefer to use an action like

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

\[ \text{(printf "Final value: \text{"a"n} x"}\]

\[ \text{(exit))} \]

instead, to really make clear what is happening.

Notice the resulting ping-pong effect between the two action values (“resumption scripts”, if you will): the first supplied by for-each/web to the user’s function, the second by the user’s function to output/web. (The third interesting Web-converted procedure in this scenario, for-each/web, provides itself with an action value, so there’s not much to hold our interest there.) The iterator provides the user’s function a complex resumption action that culminates in the recursive call. The user’s resumption action, in contrast, simply takes the junk value from the output routine, and conveys it to the iterator’s resumer.

What this demonstrates is:

- Web conversion is a pervasive process. Once one procedure is “infected”, it infects everything that it calls.
Language primitives are usually immune to infection, but not all of them. We can leave simple primitives like addition (so long as they are performed on immediate values) unchanged, but even some primitives such as `for-each`, `map`, etc., which invoke potentially infected procedures, must also convert. In general, any primitive that consumes functions (or, in Java, invokes `callbacks`) is infected.

4 Exercises

Write and Web-convert the following procedures. Assume each procedure displays the list element at the head of the list as it recurs down the list. Provide contracts for the Web-converted versions. Provide contracts for all the procedures (including the anonymous ones) in this document.

1. `length : list(\alpha) \rightarrow \text{num}`
2. `sum : list(\text{num}) \rightarrow \text{num}`
3. `remove-sym : list(\text{sym}) \text{sym} \rightarrow list(\text{sym})`
4. `map : (\alpha \rightarrow \beta) \text{list}(\alpha) \rightarrow \text{list}(\beta)`
5. `filter : (\alpha \rightarrow \text{bool}) \text{list}(\alpha) \rightarrow \text{list}(\alpha)`

You will understand this material a lot better if you hand-evaluate a few examples!

---

7Is a variable a value?