Converting Functions to Script Position

Kathi Fisler, WPI

February 24, 2004

We’ve seen that in order to write programs that behave properly as scripts, we need to move code around so that scripts start the next part of the computation before they terminate (or, put another way, so that no code depends upon a value returned from a script). The last set of notes did this intuitively. This set of notes shows you a step-by-step method for converting non-web programs to web programs.

1 Script position

A call to a script function is in script-position if the return value of the call is the return value of the expression or function that contains it. This is another way of saying that no additional computation in the program relies on the value of the call. Script-position is easiest to understand through examples. Assume we have a script called request-num:

1. In script position because the result of the call is the result of the whole expression (assuming this is the whole program).
   
   (request-num “Enter a num”)

2. Not in script position because the + uses the return value
   
   (+ (request-num “Enter a num”) 5)

3. In script position because the value of request-num is the value of the function that contains it.
   
   (define (req reqstr)
     (request-num reqstr))

4. In script position because the cond is the whole expression and the answer of a cond clause is what the cond returns.
   
   (cond [(= 5 4) (request-num “Enter a num”)]
       [else 7])

5. Not in script position because the + uses the return value of the cond.
   
   (+ (cond [(= 5 4) (request-num “Enter a num”)]
       [else 7])
   10)

6. Not in script position because the = needs the value of the request-num. The question position of a cond is never in script position.
   
   (cond [(= 5 (request-num “Enter a num”)) 9]
       [else 7])

2 Moving Script Invocations to Script Position : Examples

To transform non-web programs to versions that are safe to run on the web, we need to first locate calls to script that are not in script position and move them into script position, adjusting the rest of the code accordingly. Before we
write out the steps for doing this, let’s look at an example. Recall the age-request program from the previous set of notes. Here’s a similar version but using a more general script function to prompt for numbers:

```scheme
;;; request-num : string → number
;;; prompts user for a number
(define (request-num promptstr)
  (begin (printf promptstr)
          (read)))

;;; age-page-nonweb : → void
;;; displays ability to vote based on user’s age
(define (age-page-nonweb)
  (local ((define age (request-num "Enter your age: "))
           (cond [(>= age 18) (printf "Don’t forget to vote!" )]
                  [else (printf "You’ll be able to vote in ~a years" (~ 18 age))])))

    The call to request-age-page is not in script position in age-page-nonweb, so that call will need to move if request-age-page is to be usable as a script. Where can we move it? That call has to happen before the rest of the code executes, so let’s rip it out and move it to the front of the age-page-nonweb function, leaving a ??? marker to show where we ripped it out:

;;; age-page-nonweb : → void
;;; displays ability to vote based on user’s age
(define (age-page-nonweb)
  (request-num "Enter your age: ")

    (local ((define age ???))
           (cond [(>= age 18) (printf "Don’t forget to vote!" )]
                  [else (printf "You’ll be able to vote in ~a years" (~ 18 age))])))
Note that this isn’t valid Scheme code – we have two expressions after the define rather than one, and we have the ??? where we yanked out the old code. We can fix the ??? by turning it into a proper variable name and wrapping the code that uses that name in a lambda. We call the variable name hole since it fills a hole in the original code (where we ripped the old code out).

;;; age-page-nonweb : → void
;;; displays ability to vote based on user’s age
(define (age-page-nonweb)
  (request-num "Enter your age: ")

    (lambda (hole)
             (local ((define age hole))
                  (cond [(>= age 18) (printf "Don’t forget to vote!" )]
                         [else (printf "You’ll be able to vote in ~a years" (~ 18 age))])))
What is this lambda? This is the code that is left to run after we run request-num – the same code that we put into the submit function in the previous lecture. Recall that in that lecture the page that prompted for the number passed the value it read to submit. Here, we want to do the same thing – we send the lambda (equivalent to the old submit to request-num), and request-num passes its result to the lambda, thus plugging the “hole” with the value of the expression that we pulled out:

;;; age-page-web : → void
;;; displays ability to vote based on user’s age
(define-script (age-page-web)
  (request-num "Enter your age: 
                 (lambda (hole)
                          (local ((define age hole))

```
(cond [(>= age 18) (printf "Don’t forget to vote!")] [else (printf "You’ll be able to vote in ~a years" (~ - 18 age))]))

;; request-num : string (number → void) → void
;; prompts user for a number
(define-script (request-num promptstr action)
  (begin (printf promptstr)
          (action (read))))

We call the new parameter to request-num action to parallel the use of the “action” keyword in HTML forms (in an HTM form, the action variable holds the name of the script to call next. We’re giving a lambda instead of the name of a function for now, but that difference is minor).

Running this version of the age program produces the same behavior as the original, but now everything can run as a script (meaning that it would work on the web). Notice the new contract on request-num – rather than return a number, it now takes a function that takes a number (its old return value) and returns the same value as the age-page-web program that calls it.

3 Moving Script Invocations to Script Position : Methodology

To convert a program written with define to one that will work with define-script, follow these steps on every function fooscript that should work with define-script.

1. Change the function name and add a parameter called action (I usually rename fooscript to foo/web, just to avoid name clashes with the non-web version).

2. Wherever fooscript returns an answer, pass that answer to action. If the body of fooscript is a cond, call action on each answer in the cond.

3. Find the script call that would happen first and move it to the front of the expression (or enclosing lambda – never move a call outside of a lambda).

4. Replace the expression you removed with a new variable name (like hole). Be sure to use a different variable name each time within the same function.

5. Wrap a (lambda (hole) …) around the original expression minus the part you pulled out, and pass it as the argument to the script call you moved to the front (this will become the value for the ”action” argument).

6. Repeat from step 3 until all calls to scripts are in script position.

Remember, the final program should behave the same way as the original program, so your transformation should not change the order in which function calls or conditionals get evaluated.

Tomorrow, more examples ...