Continuations

Shriram Krishnamurthi

2001-10-12

1 Continuations

Suppose we have a computation of the form

\((+ (* 2 3) (- (/ 4 2) 5))\)

In the CPS lectures, we’ve discussed the process of determining what we do first, then what we do next, and so on. Let’s repeat that process. What do we do first? We could do either the multiplication or the division; let’s just choose the multiplication. What’s left?

\((+ \Box (- (/ 4 2) 5))\)

This isn’t really valid Scheme code because it has an unbound variable, \(\Box\), but we can make it so by binding the variable:

\((\text{lambda} (\Box) (\text{lambda} (\Box) (+ \Box (- (/ 4 2) 5))))\)

Next we have to do the division. What does that leave?

\((\text{lambda} (\Box_2) (\text{lambda} (\Box_1) (+ \Box_1 (- \Box_2 5))))\)

and so forth. These, of course, are the continuations generated by converting a program into CPS. You can also see these continuations by using DrScheme’s Stepper.

2 Continuations into Functions

As we’ve learned from the CPS lectures, we can expose the continuations in a program by converting it into CPS. That is, given a regular program, we can convert it in a methodical, automatic fashion to yield a new program (in the same source language). In so doing, we augment all procedures with an extra argument. This argument, a procedure of one value, is the continuation.

**Puzzle:** What happens when we apply the CPS transformation to a program we’ve already converted to CPS?
Our Web exercises demonstrated that the continuation is very valuable in some applications. Indeed, Web computation is all about capturing and invoking continuations (and figuring out where to store them in the meanwhile). The PLT Web server is a specialized Web server that keeps the continuations in a table in memory, associating them with special tags. When the form response mentions the tag, the server looks up the continuation in the table and invokes it. It thus greatly simplifies the act of writing Web programs. In particular, it leaves the loop structure of programs unchanged.\(^1\)

This still leaves open the question of generating the continuations. We could do this by hand-converting the programs into CPS, but this is pretty painful. We could automate this transformation, but this introduces two different problems:

1. **Debugging is difficult:** the actual running program is syntactically extremely different from the original, so the programmer would have a difficult time mapping some errors to the original.

2. **As we discussed earlier,** we need to convert *all* procedures to CPS form. This includes all libraries and third-party code. We may not even have the source for some of this code, which means we can’t perform the conversion.

Therefore, it’d be nice to have an alternative to transformation.

Fortunately, Scheme provides us such an alternative. Scheme has a syntactic construct, \texttt{let/cc}, which is a *binding* form. Its syntax is

\[
\texttt{(let/cc \langle var \rangle \langle expr \rangle)}
\]

This expression binds \texttt{var} in body. The value of \texttt{var} is a procedure of one argument. The procedure is the *continuation*\(^2\): in other words, \texttt{let/cc} automatically performs the CPS transformation! (Well, no, it doesn’t, but the effect is as if it does.)

Let’s study \texttt{let/cc} through a few examples.

1. A very simple example is

   \[
   \texttt{(let/cc \texttt{k} \texttt{3})}
   \]

   This simply evaluates to \texttt{3}. Indeed, we can write any expression in the body, and so long as it never uses the bound variable (\texttt{k} in the example above), the \texttt{let/cc} has no effect: it’s as if there was only the body.

2. Okay, so let’s use the bound variable. Consider

   \[
   \texttt{(\texttt{let/cc \texttt{k} \texttt{(k 3))}})
   \]

\(^1\)I like to say that loops in regular programs are powered entirely by electricity, whereas loops in Web programs are powered by a combination of electricity and *calories*: unless the user submits an input or confirms an output, the computation doesn’t resume, so the loop does not continue.

\(^2\)There’s one subtlety—we’ll get to it pretty soon.
The value of this program is 8. The context of the `let/cc` expression is

```
(lambda (□
  (+ 5
    □))
```

This is the value bound to `k`. Therefore, applying this to 3 gives `(+ 5 3)`, and hence 8.

Notice that we did something subtle. The continuation of the application of `(k 3)` is really

```
(lambda (□
  (+ 5
    (let/cc k □)))
```

Replacing the `□` with the application of the continuation to 3, we get

```
(+ 5
  (let/cc k
    ((lambda (□
      (+ 5
        □))
      3)))
```

which reduces to

```
(+ 5
  (let/cc k
    (+ 5 3)))
```

and then

```
(+ 5
  (let/cc k
    8))
```

This last `let/cc` expression doesn’t use `k`, so its value should be 8. Therefore this should reduce to

```
(+ 5
  8)
```

right?

In this second interpretation of `let/cc`, we’re letting the computation continue after the continuation finishes. That is, we end up performing the actions in the continuation (in this case, the addition of 5) twice. Perhaps it’s easy to see what’s happening here, but suppose you had a complicated program in which the continuation did database operations (as many Web scripts do), or launched a rocket, or something externally visible like that. We would want those operations to happen only once per use of the continuation, not twice. (Each use of the continuation—corresponding to a form submission—should add a new entry
into the logging database, but a given form submission should add only one, not two.)

To reflect this desired behavior, the continuation that Scheme creates terminates the entire program on completion. That is, the continuation bound to $k$ in the example above is

$$(\text{lambda} \ (\square))$$

$$(\text{abort} \ (\text{let/cc} \ k \ ((\text{lambda} \ (\square)) \ (\text{abort} \ (+ \ 5 \ (\square))) \ 3))))$$

(where abort terminates the entire computation). When we apply this continuation to 3, we get

$$(+ \ 5 \ (\text{let/cc} \ k \ ((\text{lambda} \ (\square)) \ (\text{abort} \ (+ \ 5 \ (\square))) \ 3))))$$

which reduces to

$$(+ \ 5 \ (\text{let/cc} \ k \ (\text{abort} \ (+ \ 5 \ 3))))$$

and then

$$(+ \ 5 \ (\text{let/cc} \ k \ (\text{abort} \ 8)))$$

which evaluates to 8. Notice that we didn’t change the rules by which let/cc evaluates, only the form of the actual continuation.

You might wonder why this abort doesn’t show up in the Web example. Actually, it does: this corresponds to the CGI program halting after each input/output. We therefore need the abort to properly capture what happens on the Web. But thinking about external events also explains why having the abort is the right solution.

3. Let’s consider a slightly more complex variation on the theme above:

$$(+ \ 5 \ (\text{let/cc} \ k \ (+ \ 10 \ (k \ 3))))$$

We should be able to determine what this evaluates to from what we’ve learned so far. The continuation bound to $k$ is the same as in the previous example
(because \texttt{let/cc} binds \texttt{k} to \textit{its} context, which hasn’t changed form the previous example to this one). The body of the \texttt{let/cc} becomes

\begin{verbatim}
(+ 10
   ((lambda (\)
      (abort
       (+ 5 \}))
    3))
\end{verbatim}

which reduces to

\begin{verbatim}
(+ 10
   (abort
    (+ 5 3)))
\end{verbatim}

Due to the presence of the \texttt{abort}, this too yields 8.

From these two examples, we can discern a few simple principles to help us understand programs that use \texttt{let/cc}. If the bound variable never sees use, then the \texttt{let/cc} has no effect at all. When the programmer does use the bound variable (call it \texttt{k}), this has the effect of evaluating the argument to \texttt{k} (simply because Scheme is an eager language), and \textit{textually replacing the entire \texttt{let/cc} expression with this value}. This is a useful textual way to determine the value of programs that use \texttt{let/cc}.\footnote{Note that this isn’t some ad hoc rule; it’s just another way of explaining contexts.} For instance, applying this rule to

\begin{verbatim}
(+ 5
   (let/cc k (k 3)))
\end{verbatim}

yields

\begin{verbatim}
(+ 5
   3)
\end{verbatim}

and hence 8; applying it to

\begin{verbatim}
(+ 5
   (let/cc k
            (+ 10 (k 3))))
\end{verbatim}

also yields

\begin{verbatim}
(+ 5
   3)
\end{verbatim}

(remember, just erase the entire \texttt{let/cc} expression, replacing it with the argument value), and hence the same answer.

If you’re used to a language like Java, this second example might look familiar. We’ve used it in a silly example for illustrative purposes, but we could put it to better use. Consider a function that multiplies a list of numbers:

\begin{verbatim}
;; prod : LON \rightarrow \textit{number}
(define (prod l)
\end{verbatim}
Suppose we feed this program the input ‘(1 2 3 0 4 5 6). It’s going to accumulate multiplications for each of the numbers until it arrives at the 0—at which point it knows that, no matter what the rest of the list, it can just return zero. Therefore, it might as well avoid these latter computations entirely. That is, we can rewrite this function as

\[
\text{(define (prod l)} \equiv\n\text{(cond (empty? l) 1)} \equiv\n\text{(cons? l)} \equiv\n\text{(cond (zero? (first l)) 0)} \equiv\n\text{(else (* (first l)} \equiv\n\text{(prod (rest l))))))}
\]

Even this solution isn’t very unsatisfactory. We accumulate several multiplications before we encounter the zero, which means we’re still performing all those (wasted) multiplications by zero on the first three numbers of the list. Put differently, the continuation when \(\text{prod}\) encounters the zero is

\[
\text{(lambda (□)} \equiv\n\text{(* 1)} \equiv\n\text{(* 2)} \equiv\n\text{(* 3)} \equiv\n\text{(□))))}
\]

which means, no matter what, we’re going to perform these pending multiplications.

Unless . . . remember that every continuation we capture using let/cc has an abort at the “bottom”. Therefore, if we set it up exactly right, we can exploit that abort to avoid doing any remaining computations. How can we do this?

Clearly, the continuation we capture has to have no multiplications in it. That is, it must look like

\[
\text{(lambda (□)} \equiv\n\text{(abort (□))}
\]

That means we can’t capture the continuation inside \(\text{prod}\): we must do so outside. This suggests splitting the program into two parts:

\[
\text{(define (prod-help l)} \equiv\n\text{(cond (empty? l) 1)} \equiv\n\text{(cons? l)}
\]
(cond
  [(zero? (first l)) 0]
  [else (* (first l)
            (prod-help (rest l)))]))

(define (prod l)
  (prod-help l))

When we first invoke prod, the continuation is “empty”. Therefore, we should capture this continuation right away:

(define (prod l)
  (let/cc k
    (prod-help l)))

Having captured it, we should send it along into prod-help:

(define (prod-help/esc l when-0)
  (cond
    [(empty? l) 1]
    [(cons? l)
      (cond
        [(zero? (first l)) 0]
        [else (* (first l)
                (prod-help/esc (rest l) when-0))])]
    )])

(define (prod l)
  (let/cc k
    (prod-help/esc l k)))

That is, we create prod-help/esc to accept a continuation (the esc is short for “escaper”, since the continuation will help us escape from the pending multiplications). When prod-help/esc recurs, it passes the continuation argument along unchanged.

At this point, we haven’t actually changed the program’s behavior any, just made it more complex. The payoff comes from actually using when-0:

(define (prod-help/esc l when-0)
  (cond
    [(empty? l) 1]
    [(cons? l)
      (cond
        [(zero? (first l)) (when-0 0)]
        [else (* (first l)
                (prod-help/esc (rest l) when-0))])]
    )])

Remember the textual replacement rule: when we invoke when-0, we textually replace the let/cc with the argument (here, 0). Thus, it’s as if whatever invoked prod was invoking the function
whenver the list contains a 0: exactly what we want. This avoids performing any multiplications at all.

**Alert:** You should think about a few subleties:

- If the list contains no zeroes, does the new `prod` behave exactly the same as the old (continuation-free) one?
- What happens if some long chain of functions invokes `prod` on a list containing a 0? Does the `abort` in the continuation cause those other functions to never complete also, just like it inhibits the multiplications in `prod`?

4. The first example explains what happens if we don’t use the bound variable anywhere, while the second and third present an immediate use of the continuation. This raises a tantalizing question: what if we don’t use the continuation right away? What if we . . . return it from the `let/cc` expression, and use it later?

Let’s consider the simplest expression of this sort:

```
(let/cc (k k))
```

This returns a continuation, namely

```
(lambda (□) (abort □))
```

That is, it’s very nearly the identity function, save for the `abort`.

Well, if it’s the identity function, then we should be able to apply it to a value and get that value back as the answer. (Indeed, the `abort` will not affect this plan in any way.) So we might try to run the program

```
((let/cc (k k))
 (25))
```

in Scheme and expect to get 25. Before you proceed further, try it out and check whether that’s what you get.

Let’s think about what happened. The continuation for the `let/cc` is

```
(lambda (□)
 (abort
 (□ 25)))
```

Notice something very important: this continuation is different from what it would be if you ran the expression stand-alone! Continuations capture everything remaining to be done, so if you change the whole program in a way that affects what remains (as we just did, above), that affects the continuation too. Returning to our program, when we apply this continuation to 25, we get

---

4Try this out in Scheme and see what Scheme prints as the response.
which of course results in the error.

This suggests the following curious challenge: to what can we apply the continuation such that the program does yield 25? Let’s think about what happens to the value we supply to the continuation: it is immediately applied to 25, and the result becomes the result of the entire computation (due to the abort). That clarifies that the value we provide the continuation should be a function of one argument that returns 25:

\[
(\text{let/cc } k k) \\
(\lambda (\text{dummy}) 25)
\]

will do the trick. (The second time around, dummy is bound to \( (\lambda (\text{dummy}) 25) \).)

Notice that even in this example, our “erasure” principle continues to hold. If we begin with the program

\[
(\text{let/cc } k k) \\
(\lambda (\text{dummy}) 25)
\]

then we eventually apply the continuation to the closure. By the code erasure principle, this converts the entire program into

\[
((\lambda (\text{dummy}) 25) \\
(\lambda (\text{dummy}) 25))
\]

and, sure enough, this results in 25.

This example illustrates three more principles. First, the continuation is just another Scheme value, meaning it can be the result of a Scheme expression—even the let/cc expression that creates it. Using it may appear to “erase” code that has already completed running; it does! Nevertheless, it’s the Scheme implementor’s problem to make sure this happens correctly.

Second, the continuation is utterly insensitive to the context of its invocation. Because the continuation terminates in an abort, no matter where it’s invoked, the computation will never return to whatever was pending at that point. Therefore, you don’t need to think about the return value of a continuation.

Third, the continuation is extremely sensitive to the context of its definition. As a result, you can’t generally take a working use of let/cc, move it into some other expression, and expect the program to behave in exactly the same way as it did before. This is true of other expressions too, of course; for instance, if you move a let expression within another one, this may alter the binding of some variables on the RHS of the let, resulting in a different value. In a let/cc, however, even this doesn’t need to happen; just moving the code changes the continuation. Therefore, you have to be very careful with programs that use continuations. It’s usually best to use them in very principled settings, such as when suspending the computation in the Web server.
5. Let’s do one more example, this too with continuations being returned outside the scope of their definition. We’ll write a little prime number generator. Since there are an infinite number of primes, we don’t want to try making a list of all of them. Instead, we’ll create a prime number “server”. Every time you want one, you just invoke a continuation which resumes the server—until it finds a prime, which it passes to your continuation. And so on.

We’ll assume you can define the procedure \texttt{prime?}. Given that procedure we’ll create one that finds the next prime. When this procedure succeeds, it wants to return two values: the prime it found, and a continuation to use when we want it to resume searching. Since functions return one value only, we’ll bundle this up in a structure:

\begin{verbatim}
(define-struct response (prime contn))
\end{verbatim}

Now we can define the “prime server”:

\begin{verbatim}
(define (next-prime previous-try return-point)
  (let ((possible-next (add1 previous-try))
        (if (prime? possible-next)
            (next-prime possible-next
             (let/cc prime-generator
              (return-point
               (make-response possible-next prime-generator)))))
            (next-prime possible-next return-point))))
\end{verbatim}

If this looks really complicated, just remember our “erasure principle”: whatever value the user passes to the continuation in the structure becomes the value of the \texttt{let/cc} expression. Provided this is another continuation, that becomes the value of \texttt{return-point} to which the next prime gets supplied.

How do we use this server? We want to initiate it on a good search value, providing it a continuation to which it supplies the answer. At the Scheme interaction loop we can write

\begin{verbatim}
> (define v1 (let/cc k (next-prime 1 k)))
\end{verbatim}

which binds \texttt{v1}:

\begin{verbatim}
> (response-prime v1) 2
> (response-contn v1) #<continuation>
\end{verbatim}

Great, we got the first prime. But we have something perhaps even more valuable: a continuation that will take us back to the prime-finding loop! What do we provide to this continuation? Well, whatever we provide must become the next value for \texttt{return-point}, which in turn gets used to convey a result back to the user. Thus, this must be another continuation:

\begin{verbatim}
> (define v2 (let/cc k ((response-contn v1) k)))
\end{verbatim}
Sure enough,

> (response-prime v2)
3
> (response-contn v2)
#<continuation>

If we repeat this process, we see the next several primes:

> (define v3 (let/cc k ((response-contn v2) k)))
> (response-prime v3)
5
> (define v4 (let/cc k ((response-contn v3) k)))
> (response-prime v4)
7
> (define v5 (let/cc k ((response-contn v4) k)))
> (response-prime v5)
11

Believe it or not, even this isn’t the most general use of continuations! However, it’ll do for a foundational understanding of programming languages. A more vexing question at this point should be how we can implement continuations. Surely we can do this using continuations in Scheme to add continuations to Rip, but this is likely to be a very unsatisfactory solution. It’ll tell us that Scheme’s continuations are very powerful, but give us no insight into how to implement continuations in a meta-language that doesn’t already have them.