Chapter 19

Programming with Continuations

For this material, please switch to the PLAI - Pretty Big language level.

In Section 18 we saw how conversion to CPS restores the Web programming interface we desire: programmers can use *web-read* and not have to "invert" the structure of the program. While in principle this accomplishes the task, in practice conversion to CPS has several disadvantages:

1. It requires access to the source of the entire program. If a procedure is defined in a library for which we don't have access to the source, or is perhaps written in a different language (as *map* often is), then the CPS translator will either fail to run or will produce potentially erroneous output (i.e., code that does not properly restore the state of the computation).

2. By replacing the machine's stack with an explicit representation in the form of receivers, it inhibits optimizations built into compilers and microprocessor architectures.

3. As we will see in Section 21.2, executing a program in CPS also assumes that the run-time system will not needlessly create stack frames (since the stack is entirely represented by the receiver). Since many languages (such as C and Java) do anyway, the program consumes memory unnecessarily. In an extreme case, a Java or C program that might have executed without exhausting memory will do after conversion into CPS.

The first of these problems is particularly compelling, since it affects not only performance but even correctness. We would benefit from an operation that automatically constructs the receiver at any point during the program's execution, instead of expecting it to have already been created through a static compilation process.

Some programming languages, notably Scheme, have such an operation. This operation creates a representation of the "rest of the computation" (which is what the receiver represented) as a procedure of one argument. Giving that procedure a value causes the remaining computation to resume with that value. This procedure is called a *continuation*. This explains where CPS obtains its name, but note that the program does not need to be transformed a priori; the continuation is created *automatically*.

Adding continuations to a language makes it easy to create a better Web programming protocol, as we shall see. But just as laziness—which was already present in the shell but was essentially an extra-lingual feature (since programmers could not explicitly control it)—, once exposed as a feature in a programming language, gave programmers immense power in numerous contexts, so do continuations. We will explore this power in greater detail.

19.1 Capturing Continuations

In Scheme, we create a value representing the continuation using one of two related constructs. The traditional form is called *call/cc*, short for "call with current continuation". *call/cc* consumes a procedure as an argument, and invokes this procedure with a continuation. That is, uses of *call/cc* typically look like

```
(call/cc
  (lambda (k)
    ··· k ···))   ;; k is the continuation
```

Because the extra **lambda** is extremely tiresome, however, Scheme provides a nicer interface to capturing the current continuation: you may instead *equivalently* write

```
(let/cc k
  ··· k ···))   ;; k is bound to the continuation
```

Note that **let/cc** is a *binding* construct: it introduces a new scope, binding the named identifier (*k*, above) in that context. For the rest of this material, we'll use **let/cc** rather than *call/cc*.[1]

19.2 Escapers

Let's write some programs using continuations. What is the value of this program?

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

We must first determine the continuation bound to k. This is the same procedure as the value of the receiver in CPS. Since in this case, there is no computation waiting to be done outside the **let/cc** expression, the receiver would be the initial receiver, namely the identity function. Therefore, this receiver is

```
(lambda (•)
  •)
```

Applying this to 3 produces the answer 3.

Consider this program:

```
(+ 1
   (let/cc k
     (k 3)))
```

[1]So why does Scheme offer *call/cc*, ghastly as it looks? Historically, the original standards writers were loath to add new binding forms, and the use of **lambda** meant they didn't need to create one. Also, *call/cc* lets us create some incredibly clever programming puzzles that we can't write quite as nicely with **let/cc** alone! Ask us for some.

What is the continuation this time? It's

```
(lambda (•)
  (+ 1 •))
```

(all the code "outside the parens"). This procedure, when invoked, yields the value 4. Because the continuation is *the rest of the computation*, we want to halt with the value 4. But this looks confusing, because substitution gives

```
  (+ 1
     (k 3)) ;; where k is bound to (lambda (•) (+ 1 •))
= (+ 1
     ((lambda (•)
        (+ 1 •))
      3))
= (+ 1
     (+ 1 3))
```

which performs the addition twice, producing the answer 5. The problem is that we're effectively applying the continuation *twice*, whereas computation should halt after it has been applied once. We will use a special notation to reflect this: **lambda**↑ will represent a procedure that, when its body finishes computing, halts the entire computation. We'll call these *escaper* procedures, for obvious reasons. That is, the continuation is really

```
(lambda↑ (•)
  (+ 1 •))
```

so the expression

```
(+ 1
   ((lambda↑ (•)
      (+ 1 •))
    3))
```

evaluates to 4, with the outermost addition ignored (because we invoked an escaper).

19.3 Exceptions

Let's consider a similar, but slightly more involved, example. Suppose we are deep in the midst of a computation when we realize we are about to divide by zero. At that point, we realize that we want the value of the entire expression to be one. We can use continuations to represent this pattern of code:

```
(define (f n)
  (+ 10
     (* 5
        (let/cc k
          (/ 1 n)))))
(+ 3 (f 0))
```

The continuation bound to *k* is

```
(lambda↑ (•)
  (+ 3
     (+ 10
        (* 5
           •))))
```

but oops, we're about to divide by zero! Instead, we want the entire division expression to evaluate to one. Here's how we can do it:

```
(define (f n)
  (+ 10
     (* 5
        (let/cc k
          (/ 1 (if (zero? n)
                   (k 1)
                   n))))))
```

so that • in the continuation is substituted with 1, we bypass the division entirely, and the program can continue to evaluate.

Have you seen such a pattern of programming before? But of course: *k* here is acting as an *exception handler*, and the *invocation* of *k* is *raising the exception.* A better name for *k* might be *esc*:

```
(define (f n)
  (+ 10
     (* 5
        (let/cc esc
          (/ 1 (if (zero? n)
                   (esc 1)
                   n))))))
```

which makes pretty clear what's happening: *when you invoke the continuation*, it's as if the entire **let/cc** expression that binds *esc* should be cut out of the program and replaced with the value passed to *esc*, i.e., its as if the actual code for *f* is really this:

```
(define (f n)
  (+ 10
     (* 5
        1)))
```

In general, this "cut-and-paste" semantics for continuations is the simplest way (in conjunction with escaper procedures) of understanding a program that uses continuations.

There was, incidentally, something sneaky about the program above: it featured an expression in the body of a **let/cc** that did *not* invoke the continuation. That is, if you can be sure the user of *f* will never pass an argument of 0, it's as if the body of the procedure is really

```
(define (f n)
(+ 10
   (* 5
      (let/cc esc
         (/ 1 n)))))
```

but we haven't talked about what happens in programs that don't invoke the continuation. In fact, that's quite easy: the value of the entire **let/cc** expression is exactly that of the value of its body, just as when you don't actually raise an exception.

19.4 Web Programming

Now that we're getting a handle on continuations, it's easy to see how they apply to the Web. We no longer need the procedure *web-read/k*; now, *web-read* can be implemented directly to do the same thing that *web-read/k* was expected to do. *web-read* captures the current continuation, which corresponds to the second argument supplied to *web-read/k* (except the continuation is now captured automatically, instead of having to be supplied as an explicit second argument).

The rest of the implementation is just as before: it stores the continuation in a fresh hash table entry, and generates a URL containing that hash table entry's key. The launcher extracts the continuation from the hash table and applies it to the user's input. As a result, all the programs we have written using *web-read* are now directly executable, without the need for the CPS transformation.

19.5 Producers and Consumers

A number of programs follow a *producer-consumer* metaphor: one process generates (possibly an infinite number of) values, while another consumes them as it needs new ones. We saw several examples of this form in Haskell, for instance. Many client-server programs are like this. Web programs have this form (we, the user, are the supplier of inputs—and on some Web sites, the number really does seem quite close to infinite ...). I/O, in general, works this way. So it's worth understanding these processes at a deeper level.

To avoid wrangling with the complexities of these APIs, we'll reduce this to a simpler problem. We'd like to define a producer of numbers that takes one argument, *send*, which masks the details of the API, and sends a sequence of numbers on demand:

```
(define (number-producer send)
  (begin
    (send 1)
    (send 2)
    (send 3)))
```

That is, when we first invoke *number-producer*, it invokes *send* with the value 1—and halts. When we invoke it again, it sends 2. The third time we invoke it, it sends the value 3. (For now, let's not worry about what happens if we invoke it additional times.)

What do we supply as a parameter to elicit this behavior? Clearly, we can't simply supply an argument such as (**lambda** (*x*) *x*). This would cause all three *send* operations to happen in succession, returning the value of the third one, namely 3. Not only is this the wrong value on the first invocation, it also produces the same answer no matter how many times we invoke *number-producer*—no good.

In fact, if we want to somehow suspend the computation, it's clear we need one of these exception-like operations so that the computation halts prematurely. So a simple thing to try would be this:

(**let/cc** *k* (*number-producer k*))

What does this do? The continuation bound to *k* is

(**lambda**↑ (•) •)

Substituting it in the body results in the following program:

```
(begin
   ((lambda↑ (•) •)
    1)
   ((lambda↑ (•) •)
    2)
   ((lambda↑ (•) •)
    3))
```

(all we've done is substitute *send* three times). When we evaluate the first escaper, the entire computation reduces to 1—and computation halts! In other words, we see the following interaction:

```
> (let/cc k (number-producer k))
1
```

This is great—we've managed to get the program to suspend after the first *send*! So let's try doing this a few more times:

```
> (let/cc k (number-producer k))
1
> (let/cc k (number-producer k))
1
> (let/cc k (number-producer k))
1
```

Hmm—that should dampen our euphoria a little.

What's the problem? We really want *number-producer* to "remember" what value it last sent, so it can send us the next value when we invoke it again. Well, actually, that's a slightly lame way of thinking about it. If producers only generate sequences of natural numbers, this would suffice, but obviously most interesting computations do a lot more than that: they generate names, addresses, credit card numbers, even other procedures ... What we *really* mean is that we want to remember *what we last did*, and then resume from where we left off *in the computation*. That's a much more general way of thinking about this, because it's independent of the kind of values we're generating! (Indeed, the whole point of the producer may not be to compute values so much as to perform actions—which this reformulation captures just as gracefully.) But to resume, we must first capture a snapshot of our computation before we sent a value. That sounds familiar ...

Going back to *number-producer*, let's think about what the continuation is at the point of invoking *send* the first time. The continuation is

```
(lambda↑ (●)
  (begin
    ●
    (send 2)
    (send 3)))
```

(with *send* substituted appropriately). Well, this looks promising! If we could somehow hang on to this continuation, we could use it to resume the producer by sending 2, and so forth.

To hang on to the computation, we need to store it somewhere (say in a box), and invoke it when we run the procedure the next time. What is the initial value to place in the box? Well, initially we don't know what the continuation is going to be, and anyway we don't need to worry because we know how to get a value out the first time (we just saw how, above). So we'll make the box initially contain a flag value, such as false. Examining what's in the box will let us decide whether we're coming through the first time or not. If we are, we proceed as before, otherwise we need to capture continuations and what-not.

Based on this, we can already tell that the procedure is going to look roughly like this:

```
(define number-producer
  (local ([define resume (box false)])
    (lambda (send)
      (if (unbox resume)
          ;; then
          there's a continuation present—do something!
          ;; else
          (begin
            (send 1)
            (send 2)
            (send 3))))))
```

where the box bound to *resume* stores the continuation. Note that *resume* is outside the **lambda**, but in its scope, so the identifier is defined only once, and all invocations of the procedure can share it.

If (*unbox resume*) doesn't evaluate to false,[2] that means the box contains a continuation. What can we do with continuations? Well, really only one thing: invoke them. So we must have something like this if the test succeeds:

```
((unbox resume) ···)
```

But what is that continuation? It's one that goes into the midst of the **begin** (recall we wrote it down above). Since we really don't care about the value, we may as well pass the continuation some kind of dummy value. Better to pass something like 'dummy, which can't be confused for any real value produced by this computation (such as a number). So the procedure now looks like

```
(define number-producer
```

[2]In Scheme, only false fails a conditional test; all other values succeed.

```
  (local ([define resume (box false)])
    (lambda (send)
      (if (unbox resume)
          ((unbox resume) 'dummy)
          (begin
            (send 1)
            (send 2)
            (send 3))))))
```

Of course, we haven't actually done any of the hard work yet. Recall that we said we want to capture the continuation of *send*, but because *send* is given as a parameter by the user, we can't be sure it'll do the right thing. Instead, we'll locally define a version of *send* that does the right thing. That is, we'll rename the *send* given by the user to *real-send*, and define a *send* of our own that invokes *real-send*.

What does our *send* need to do? Well, obviously it needs to capture its continuation. Thus:

```
(define number-producer
  (local ([define resume (box false)])
    (lambda (real-send)
      (local ([define send (lambda (value-to-send)
                             (let/cc k
                               ···))])
        (if (unbox resume)
            ((unbox resume) 'dummy)
            (begin
              (send 1)
              (send 2)
              (send 3)))))))
```

What do we do with *k*? It's the continuation that we want to store in *resume*:

```
(set-box! resume k)
```

But we also want to pass a value off to *real-send*:

```
(real-send value-to-send)
```

So we just want to do this in sequence!

```
(begin
  (set-box! resume k)
  (real-send value-to-send))
```

When the client next invokes *number-producer*, the continuation stored in the box bound to *resume* is that of invoking (our locally defined) *send* within the body ... which is exactly where we want to resume computation! Here's our entire procedure:

```
(define number-producer
  (local ([define resume (box false)])
    (lambda (real-send)
      (local ([define send (lambda (value-to-send)
                             (let/cc k
                               (begin
                                 (set-box! resume k)
                                 (real-send value-to-send))))])
        (if (unbox resume)
            ((unbox resume) 'dummy)
            (begin
              (send 1)
              (send 2)
              (send 3)))))))
```

Here's the interaction we wanted:[3]

```
> (let/cc k (number-producer k))
1
> (let/cc k (number-producer k))
2
> (let/cc k (number-producer k))
3
```

It's a bit unwieldy to keep writing these **let/cc**s, so we can make our lives easier as follows:

```
  (define (get producer)
    (let/cc k (producer k)))
```

(we could equivalently just define this as (**define** *get call/cc*)) so that

```
> (get number-producer)
1
> (get number-producer)
2
> (get number-producer)
3
```

Exercise 19.5.1 *How would you define the same operators in Haskell?*

Exercise 19.5.2 *Before you read further: do you see the subtle bug lurking in the definitions above?*
Hint: *We would expect the three invocations of (get number-producer) above, added together, to yield* 6.

[3]It's critical that you work through the actual continuations by hand.

Continuations and "Goto" Statements

What we're doing here has already gone far beyond the simple exception pattern we saw earlier. There, we used continuations only to ignore partial computations. Now we're doing something much richer. We're first executing part of a computation in a producer. At some point, we're binding a continuation and storing it in a persistent data structure. Then we switch to performing some completely different computation (in this case, the top-level requests for the next number). At this point, the partial computation of the producer has seemingly gone away, because we invoked an escaper to return to the top-level. But by accessing the continuation in the data structure, we are able to *resume a prior computation*: that is, we can not only jump "out", as in exceptions, but we can even jump back "in"! We do "jump back in" in the Web computations, but that's a much tamer form of resumption. What we're doing here is resuming between two separate computations!

All this should raise the worrying specter of "goto" statements. Let's examine that comparison a little.

Goto statements can usually jump to an arbitrary line that may not even have its variables initialized, and have other nonsensical state. In contrast, a continuation only allows you to resume a computational state you have visited before, so if you leave things in a coherent state, they should be coherent when you resume. By being more controlled in that sense, continuations avoid the worst perils of gotos.

On the other hand, continuations are far more powerful than typical goto statements. Usually, a goto statement can only transfer control to a lexically proximate statement (due to how compilers work). In contrast, a computation can represent any arbitrary prior computation and, irrespective of lexical proximity, the programmer can resurrect this computation.

In short, continuations are more structured, yet much more powerful, than goto statements. The warnings about unfettered use of goto statements apply to continuations also. One redeeming element is that continuations can be sufficiently tricky for novices that most novices wouldn't dare use them (at least not for long). And as are seeing, they do provide incredibly expressive power to skilled programmers.

19.6 A Better Producer

The producer shown above is pretty neat—indeed, we'd like to be able to use this in more complex computations. For instance, we'd like to initialize the producer, then write

```
(+ (get number-producer)
   (get number-producer)
   (get number-producer))
```

Evaluating this in DrScheme produces... *an infinite loop*!

What went wrong here? It's revealing to use this version of the program instead:

```
(+ (let/cc k (number-producer k))
   (let/cc k (number-producer k))
   (let/cc k (number-producer k)))
```

After a while, click on Break in DrScheme. If you try it a few times, sooner or later you'll find that the break happens at the *second* **let/cc**—but never the third! In fact, if you ran this program instead:

```
(+ (let/cc k (number-producer k))
   (begin
     (printf "got here!~n")
     (let/cc k (number-producer k)))
   (begin
     (printf "here too!~n")
     (let/cc k (number-producer k))))
```

you'd find several got heres in the in the Interactions window, but no here toos. That's revealing!

What's going on? Let's analyze this carefully. The first continuation bound to *real-send* in *number-producer* is

```
(lambda↑ (•)
  (+ •
     (let/cc k (number-producer k))
     (let/cc k (number-producer k))))
```

Since we are invoking *number-producer* for the first time, in its body we eventually invoke *send* on 1. What's the computation that we capture and store in *resume*? Let's compute this step-by-step. The top-level computation is

```
(+ (let/cc k (number-producer k))
   (let/cc k (number-producer k))
   (let/cc k (number-producer k)))
```

Substituting the body of *number-producer* in place of the first invocation, and evaluating the conditional, we get

```
(+ (begin
     (send 1)
     (send 2)
     (send 3))
   (let/cc k (number-producer k))
   (let/cc k (number-producer k)))
```

We're playing fast-and-loose with scope—technically, we should write the entire **local** that binds *send* between the first **let/cc** and **begin**, as well as all the values bound in the closure, but let's be a bit sloppy for the sake of readability. Just remember what *real-send* is bound to—it'll be relevant in a little while!

When *send* now captures its continuation, what it really captures is

```
(lambda↑ (•)
  (+ (begin
       •
       (send 2)
       (send 3))
```

```
      (let/cc k (number-producer k))
      (let/cc k (number-producer k))))
```

send stores this continuation in the box, then supplies 1 to the continuation passed as *real-send*. This reduces the entire computation to

```
(+ 1
   (let/cc k (number-producer k))
   (let/cc k (number-producer k)))
```

The second continuation therefore becomes

```
(lambda↑ (•)
  (+ 1
     •
     (let/cc k (number-producer k))))
```

which becomes the new value of *real-send*. The second time into *number-producer*, however, we do have a value in the box bound to *resume*, so we have to extract and invoke it. We do, resulting in

```
((lambda↑ (•)
   (+ (begin
        •
        (send 2)
        (send 3))
      (let/cc k (number-producer k))
      (let/cc k (number-producer k))))
 'dummy)
```

which makes

```
(+ (begin
     'dummy
     (send 2)
     (send 3))
   (let/cc k (number-producer k))
   (let/cc k (number-producer k))))
```

the entire computation (because we invoked a **lambda↑**). This looks like it should work fine! When we invoke *send* on 2, we capture the second computation, and march down the line.

Unfortunately, a very subtle bug is lurking here. The problem is that the *send* captured in the continuation is closed over the *old* value of *real-send*, because the *send* that the continuation closes over is from the previous invocation of *number-producer*! Executing the **begin** inside *send* first stores the new continuation in the box bound to *resume*:

```
(lambda↑ (•)
  (+ (begin
       •
```

```
        (send 3))
      (let/cc k (number-producer k))
      (let/cc k (number-producer k))))
```

but then executes the old *real-send*:

```
((lambda↑ (•)
   (+ •
      (let/cc k (number-producer k))
      (let/cc k (number-producer k))))
 2)
```

In other words, the entire computation now becomes

```
(+ 2
   (let/cc k (number-producer k))
   (let/cc k (number-producer k)))
```

which is basically the same thing we had before! (We've got 2 instead of 1, but the important part is what follows it.) As you can see, we're now stuck in a vicious cycle. Now we see why it's the *second* sub-expression in the sum where the user break occurs—it's the one that keeps happening over and over (the first is over quickly, while the computation never gets to the third!).

This analysis gives us a good idea of what's going wrong. Even though we're passing in a fresh, correct value for *real-send*, the closure still holds the old and, by now, wrong value. We need the new value to somehow replace the value in the old closure.

This sounds like the task of an assignment statement, and that's certainly an easy way to accomplish this. What we want to mutate is the value bound to *real-send*. So we'll first put it in a box and refer to the value in that box:

```
(define number-producer
  (local ([define resume (box false)])
    (lambda (real-send)
      (local ([define send-to (box real-send)]
              [define send (lambda (value-to-send)
                             (let/cc k
                               (begin
                                 (set-box! resume k)
                                 ((unbox send-to) value-to-send))))])
        (if (unbox resume)
            ((unbox resume) 'dummy)
            (begin
              (send 1)
              (send 2)
              (send 3)))))))
```

So far, this hasn't actually fixed anything. However, we now have a box whose contents we can replace. We should replace it with the new value for *real-send*. Where is that new value available? It's available at the time of invoking the continuation—that is, we could pass that new value along instead of passing 'dummy. Where does this value go? Using our "replacing text" semantics, it replaces the entire (**let/cc** ···) expression in the definition of *send*. Therefore, that expression evaluates to the new value to be put into the box. All we need to do is actually update the box:

```
(define number-producer
  (local ([define resume (box false)])
    (lambda (real-send)
      (local ([define send-to (box real-send)]
              [define send (lambda (value-to-send)
                             (set-box! send-to
                                       (let/cc k
                                         (begin
                                           (set-box! resume k)
                                           ((unbox send-to) value-to-send)))))])
        (if (unbox resume)
            ((unbox resume) real-send)
            (begin
              (send 1)
              (send 2)
              (send 3)))))))
```

This time, even though *send* invokes the old closure, the box in that closure's environment now has the continuation for the new resumption point. Therefore,

```
> (+ (get number-producer)
     (get number-producer)
     (get number-producer))
6
```

There's just one problem left with this code: it deeply intertwines two very separate concerns, one of which is sending out the actual values (which is specific to whatever domain we are computing in) and the other of which is erecting and managing the continuation scaffold. It would be nice to separate these two.

This is actually a lot easier than it looks. Simply factor out the body into a separate procedure:

```
(define (number-producer-body send)
  (begin
    (send 1)
    (send 2)
    (send 3)))
```

(This, you recall, is what we set out to write in the first place!) Everything else remains in a general procedure that simply consumes (what used to be) "the body":

```
(define (general-producer body)
  (local ([define resume (box #f)])
    (lambda (real-send)
      (local ([define send-to (box real-send)]
              [define send (lambda (value-to-send)
                             (set-box! send-to
                                       (let/cc k
                                         (begin
                                           (set-box! resume k)
                                           ((unbox send-to) value-to-send)))))])
        (if (unbox resume)
            ((unbox resume) real-send)
            (body send))))))
```

And that's it! Introducing this separation makes it easy to write a host of other value generators. They can even generate a potentially infinite number of values! For instance, here's one that generates the odd positive intgers:

```
(define (odds-producer-body send)
  (local ([define (loop n)
            (begin
              (send n)
              (loop (+ n 2)))])
    (loop 1)))
```

We can make an actual generator out of this as follows:

```
(define odds-producer (general-producer odds-producer-body))
```

and then invoke it in a number of different ways:

```
> (get odds-producer)
1
> (get odds-producer)
3
> (get odds-producer)
5
> (get odds-producer)
7
> (get odds-producer)
9
```

or (assuming an initialized program)

```
> (+ (get odds-producer)
     (get odds-producer)
     (get odds-producer)
     (get odds-producer)
     (get odds-producer))
25
```

19.7 Why Continuations Matter

Continuations are valuable because they enable programmers to create new control operators. That is, if a language did not already have (say) a simple producer-consumer construct, continuations make it possible for the programmer to build them manually and concisely. More importantly, these additions can be *encapsulated* in a library, so the rest of the program does not need to be aware of them and can use them as if they were built into the language. If the language did not offer continuations, some of these operations would require a whole-program conversion into CPS. This would not only place an onerous burden on the users of these operations, in some cases this may not even be possible (for instance, when some of the source is not available). This is, of course, the same argument we made for not wanting to rely on CPS for the Web, but the other examples illustrate that this argument applies in several other contexts as well.