Chapter 20

Implementing Continuations

Now that we’ve seen how continuations work, let’s study how to implement them in an interpreter.

The first thing we’ll do is change our representation of closures. Instead of using a structure to hold the pieces, we’ll use Scheme procedures instead. This will make the rest of this implementation a lot easier.\(^1\)

First, we modify the datatype of values. Only two rules in the interpreter need to change: that which creates procedure values and those that consume them. These are the \texttt{fun} and \texttt{app} cases, respectively. Both rules are very straightforward: one creates a closure and wraps it in an instance of \texttt{closureV}, while the other extracts closure in the \texttt{closureV} structure and applies it to the argument. The code is in Figure 7.1. Note that this change is so easy only because functions in the interpreted language closely match those of Scheme: both are eager, and both obey static scope. (As we’ve discussed before, this is our the usual benefit of using meta interpreters that match the interpreted language.)

20.1 Representing Continuations

To implement continuations in the interpreted language, we must first make them explicit in the interpreter. We do that, as you may have guessed, by converting the interpreter into \texttt{CPS}. What follows, therefore, is a blow-by-blow account of converting the interpreter. We suggest that you, dear reader, first conduct this experiment on your own, then read the discussion to confirm that you converted it properly. Look out: there is one subtlety!

We will begin by making continuations explicit in the interpreter. This is subtly different from adding continuations to the language being interpreted—we’ll do that in the next section. For now, we just want to determine what the continuation is at each point. To do this, we’ll need to transform the interpreter.

We’ll assume that the interpreter takes an extra argument \(k\), a receiver. The receiver expects the answer from each expression’s interpretation. Thus, if the interpreter already has a value handy, it supplies that value to the receiver, otherwise it passes a (possibly augmented) receiver along to eventually receive the value of that expression. The cardinal rule is this: \textit{We never want to use an invocation of interp as a sub-expression of some bigger expression}. Instead, we want interp to communicate its answer by passing it to the given

\(^1\)Yes, we’re switching to a more meta interpreter, but this is acceptable for two reasons: (1) by now, we understand procedures extremely well, and (2) the purpose of this lecture is to implement continuations, and so long as we accomplish this without using Scheme continuations, we won’t have cheated.
$k$. (Think of the interpreter as executing remotely over the Web. That is, each time we invoke \textit{interp}, the computation is going to halt entirely; only the receiver gets stored on the server. If we fail to bundle any pending computation into the receiver, it’ll be lost forever."

Let’s consider the simplest case, namely numbers. A number needs no further evaluation: it is already a value. Therefore, we can feed the number to the awaiting receiver.

\[
\text{num } (n) \ (k \ (\text{numV } n))\]

Identifiers and closures, already being values, look equally easy:

\[
\text{id } (v) \ (k \ (\text{lookup } v \ sc))\]

\[
\text{fun } (\text{param body}) \\
(k \ (\text{closureV } (\textbf{lambda} \ (\text{arg-val}) \\
(\text{interp body } (aSub \ \text{param } \text{arg-val} \ sc))))))\]

Now let’s tackle addition. The rule currently looks like this:

\[
\text{add } (l \ r) \ (\text{numV}+ \ (\text{interp } l \ sc) \ (\text{interp } r \ sc))\]

The naive solution might be to transform it as follows:

\[
\text{add } (l \ r) \ (k \ (\text{numV}+ \ (\text{interp } l \ sc) \ (\text{interp } r \ sc)))\]

but do we have a value immediately handy to pass off to $k$? We will after interpreting both sub-expressions and adding their result, but we don’t just yet. Recall that we can’t invoke \textit{interp} in the midst of some larger computation. Therefore, we need to bundle that remaining computation into a receiver. What is that remaining computation?

We can calculate the remaining computation as follows. In the naive version, what’s the first thing the interpreter needs to do? It must evaluate the left sub-expression\footnote{Notice that once again, we’ve been forced to choose an order of evaluation, just as we had to do to implement state.}. So we write that first, and move all the remaining computation into the receiver of that invocation of the interpreter:

\[
\text{add } (l \ r) \ (\text{interp } l \ sc \\
(\textbf{lambda} \ (lv) \\
(k \ (\text{num}+ \ lv \ (\text{interp } r \ sc))))))\]

In other words, in the new receiver, we record the computation waiting to complete after reducing the left sub-expression to a value. However, this receiver is not quite right either. It has two problems: the invocation of \textit{interp} on $r$ has the wrong arity (it supplies only two arguments, while the interpreter now consumes three), and we still have an invocation of the interpreter in a sub-expression position. We can eliminate both problems by performing the same transformation again:

\[
\text{add } (l \ r) \ (\text{interp } l \ sc \\
(\textbf{lambda} \ (lv) \\
(\text{interp } r \ sc \\
(\textbf{lambda} \ (rv) \\
(k \ (\text{num}+ \ lv \ rv)))))))\]
That is, the first thing to do in the receiver of the value of the left sub-expression is to interpret the right sub-expression; the first thing to do with its value is to add them, and so on.

Can we stop transforming now? It is true that interp is no longer in a sub-expression—it’s always the first thing that happens in a receiver. What about the invocation of numV+, though? Do we have to transform it the same way?

It depends. When we perform this transformation, we have to decide which procedures are primitive and which ones are not. The interpreter clearly isn’t. Usually, we treat simple, built-in procedures such as arithmetic operators as primitive, so that’s what we’ll do here (since num+ is just a wrapper around addition). Had we chosen to transform its invocation also, we’d have to add another argument to it, and so on. As an exercise, you should consider how the resulting code would look.

Now let’s tackle the conditional. Clearly the interpretation of the test expression takes place in a sub-expression position, so we’ll need to lift it out. An initial transformation would yield this:

\[
\text{if0 (test pass fail)}
\text{ (interp test sc)}
\text{ (lambda (tv))}
\text{ (if (num-zero? tv)}
\text{ (interp pass sc ???))}
\text{ (interp fail sc ???))})
\]

Do we need to transform the subsequent invocations of interp? No we don’t! Once we perform the test, we interpret one branch or the other, but no code in this rule is awaiting the result of interpretation to perform any further computation—the result of the rule is the same as the result of interpreting the chosen branch.

Okay, so what receivers do they use? The computation they should invoke is the same computation that was awaiting the result of evaluating the conditional. The receiver k represents exactly this computation. Therefore, we can replace both sets of ellipses with k:

\[
\text{if0 (test pass fail)}
\text{ (interp test sc)}
\text{ (lambda (tv))}
\text{ (if (num-zero? tv)}
\text{ (interp pass sc k))}
\text{ (interp fail sc k))})
\]

That leaves only the rule for application. The first few lines of the transformed version will look familiar, since we applied the same transformation in the case of addition:

\[
\text{app (fun-expr arg-expr)}
\text{ (interp fun-expr sc)}
\text{ (lambda (fun-val)}
\text{ (interp arg-expr sc)}
\text{ (lambda (arg-val))}
\]

3 Using our Web analogy, the question is which primitives might arguably invoke a Web interaction. Ones that use the Web must be transformed and be given receivers to stash on the server, while ones that don’t can remain unmolested. Arithmetic, clearly, computes entirely locally.
All we have to determine is what to write in place of the box.

What was in place of the box was \((\text{closureV-p } \text{fun-val}) \text{ arg-val}\). Is this still valid? Well, the reason we write interpreters is so that we can experiment! How about we just try it on a few expressions and see what happens?

```
> (interp-test '5 5)
#t
> (interp-test '+ 5 5 10)
#t
> (interp-test '{with {x (+ 5 5)} (+ x x)} 20)
procedure interp: expects 3 arguments, given 2 ...
```

Oops! DrScheme highlights the interpretation of the body in the rule for \(\text{fun}\).

Well, of course! The interpreter expects three arguments, and we’re supplying it only two. What should the third argument be? It needs to be a receiver, but which one? In fact, it has to be whatever receiver is active at the time of the procedure invocation. This is eerily reminiscent of the store: while the environment stays static, we have to pass this extra value that reflects the current state of the dynamic computation. That is, we really want the rule for functions to read

\[
\begin{align*}
\text{[fun (param body)} & \\
& \quad (k (\text{closureV} (\lambda (\text{arg-val dyn-k)}) \\
& \quad \quad \quad \text{interp body (aSub param arg-val sc) dyn-k}))])
\end{align*}
\]

(What happens if we use \(k\) instead of \(\text{dyn-k}\) in the invocation of the interpreter? Try it and find out!)

Correspondingly, application becomes

\[
\begin{align*}
\text{[app (fun-expr arg-expr)} & \\
& \quad \text{(interp fun-expr sc} \\
& \quad \quad (\lambda (\text{fun-val}) \\
& \quad \quad \quad \text{(interp arg-expr sc} \\
& \quad \quad \quad \quad (\lambda (\text{arg-val}) \\
& \quad \quad \quad \quad \quad ((\text{closureV-p } \text{fun-val}) \\
& \quad \quad \quad \quad \quad \text{arg-val k})))))]
\end{align*}
\]

The core of the interpreter is in Figure 20.1.

What, incidentally, is the type of the interpreter? Obviously it now has one extra argument. More interestingly, what is its return type? It used to return values, but now... it doesn’t return! That’s right: whenever the interpreter has a value, it passes the value off to a receiver.

### 20.2 Adding Continuations to the Language

At this point, we have most of the machinery we need to add continuations explicitly as values in the language. The receivers we have been implementing are quite similar to the actual continuations we need. They appear to differ in two ways:
1. They capture what’s left to be done in the interpreter, not in the user’s program.

2. They are regular Scheme procedures, not \texttt{lambda} procedures.

However,

1. \textit{They capture what’s left to be done in the interpreter, not in the user’s program.} Because the interpreter closes over the expressions of the user’s program, invocations of the interpreter simulate the execution of the user’s program. Therefore, the receivers effectively do capture the continuations of the user’s program.

2. \textit{They are regular Scheme procedures, not \texttt{lambda} procedures.} We have taken care of this through the judicious use of a programming pattern. Recall our discussion of the type of the revised interpreter? The interpreter never returns—thereby effectively making the receiver behave like an escaper!

In other words, we’ve very carefully set up the interpreter to truly represent the continuations, making a continuation-capturing primitive very easy to implement.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Adding Continuations to Languages} & \\
\hline
Different languages take different approaches to adding continuations. Scheme’s is the most spartan. It add just one primitive procedure, \texttt{call/cc}. The resulting continuations can be treated as if they were procedures, so that procedure application does double duty. DrScheme slightly enriches the language by also providing \texttt{let/cc}, which is a binding construct, but it continues to overload procedure application.
\end{tabular}
\end{table}

The functional language SML uses \texttt{callcc} (which is not a binding construct) to capture continuations, but adds a \texttt{throw} construct to invoke continuations. Consequently, in SML, procedure application invokes only procedures, and \texttt{throw} invokes only continuations, making it possible for a type-checker to distinguish between the two cases.

It’s possible that a language could have both a binding construct like \texttt{let/cc} and a separate \texttt{throw}-like construct for continuation invocation, but there don’t appear to be any.

In a way, the traditional Scheme approach of providing only \texttt{call/cc} is insidious. Normally, procedural primitives such as + are extremely simple, often implementable directly in terms of a small number of standard machine code instructions. In contrast, continuation capture masquerades as a procedural primitive, but it significantly changes the language semantics. This is arguably a bad design decision, because it fails to provide the student of the language a signpost of an impending “dangerous bend”.

To implement continuations, we will take the DrScheme approach of adding a binding construct but overloading procedural application:

\[
\begin{align*}
\texttt{<KCFAE>} & ::= \texttt{<num>} \\
 & \quad | \{+ \texttt{<KCFAE>} \texttt{<KCFAE>}\} \\
 & \quad | \{\texttt{if0} \texttt{<KCFAE>} \texttt{<KCFAE>} \texttt{<KCFAE>}\} \\
 & \quad | \texttt{<id>} 
\end{align*}
\]
CHAPTER 20. IMPLEMENTING CONTINUATIONS

| {fun {<id>} <KCFAE>} |
| {<KCFAE> <KCFAE>} |
| {withcc <id> <KCFAE>} |

(This is just for purposes of illustration. If you would prefer to have an explicit continuation invocation construct such as throw, you could easily add it with a handful of lines of very straightforward code.) The revised language is thus as follows:

(define-type KCFAE

  [num (n number?)]
  [add (lhs KCFAE?) (rhs KCFAE?)]
  [if0 (test KCFAE?) (pass KCFAE?) (fail KCFAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body KCFAE?)]
  [app (fun-exp KCFAE?) (arg-exp KCFAE?)]
  [withcc (cont-var symbol?) (body KCFAE?)])

We need to add one new rule to the interpreter, and update the existing rule for application. We’ll also add a new kind of type, called contV.

How does bindcc evaluate? Clearly, we must interpret the body in an extended environment:

[bindcc (cont-var body)
  (interp body
   (aSub cont-var
    (contV ???)
    sc)
   k)]

The receiver used for the body is the same as the receiver of the bindcc expression. This captures the intended behavior when the continuation is not used: namely, that evaluation proceeds as if the continuation were never captured.

What kind of value should represent a continuation? Clearly it needs to be a Scheme procedure, so we can apply it later. Functions are represented by procedures of two values: the parameter and the continuation of the application. Clearly a continuation must also take the value of the parameter. However, the whole point of having continuations in the language is to ignore the continuation at the point of invocation and instead employ the one stored in the continuation value. Therefore, it would make no sense to accept the application’s continuation as an argument, since we’re going to ignore it anyway. Instead, the continuation uses that captured at the creation, not use, of the continuation:

[bindcc (cont-var body)
  (interp body
   (aSub cont-var
    (contV (lambda (val)
             (k val))
    sc)
   k)]
(Note again the reliance on Scheme’s static scope to close over the value of \( k \).) This makes the modification to the application clause very easy:

\[
\begin{align*}
\text{app} & (\text{fun-expr} \ \text{arg-expr}) \\
& (\text{interp fun-expr sc} \\
& \quad (\lambda (\text{fun-val}) \\
& \quad \quad (\text{interp arg-expr sc} \\
& \quad \quad \quad (\lambda (\text{arg-val}) \\
& \quad \quad \quad \quad \text{(type-case KCFAE-Value fun-val} \\\n& \quad \quad \quad \quad \quad \quad \text{[closureV (c) (c arg-val k)]} \\\n& \quad \quad \quad \quad \quad \quad \quad \text{[contV (c) (c arg-val)]} \\\n& \quad \quad \quad \quad \quad \quad \text{[else (error "not an applicable value")])})]) \\
\end{align*}
\]

Notice the very clear contrast between function and continuation application: function application employs the receiver at the point of application, whereas continuation application employs the receiver at the point of creation. This difference is dramatically highlighted by this code.

One last matter: what is the initial value of \( k \)? If we want to be utterly pedantic, it should be all the computation we want to perform with the result of interpretation—i.e., a representation of “the rest of DrScheme”. In practice, it’s perfectly okay to use the identity function. Then, when the interpreter finishes its task, it invokes the identity function, which returns the value to DrScheme.

And that’s it! In these few lines, we have captured the essence of the meaning of continuations. (The heart of the interpreter is in Figure 20.2.) Note in particular two properties of continuations that are captured by, but perhaps not obvious from, this implementation:

- To reiterate: we ignore the continuation at the point of application, and instead use the continuation from the point of creation. This is the semantic representation of the intuition we gave earlier for understanding continuation programs: “replace the entire \textit{let/cc} expression with the value supplied to the continuation”. Note, however, that the captured continuation is itself a dynamic object—it depends on the entire history of calls—and thus cannot be computed purely from the program source without evaluation. In this sense, it is different from the environment in a closure, which can partially be determined entirely statically (that is, we can determine which identifiers are in the environment, though it is undecidable what their values will be.)

- The continuation closes over the environment; in particular, its body is scoped statically, not dynamically.

### 20.3 Testing

You might think, from last time’s extended example of continuation use, that it’s absolutely necessary to have state to write any interesting continuation programs. While it’s true that most practical uses of the full power of continuations (as opposed to merely exceptions, say) do use state, it’s possible to write some fairly complicated continuation programs without state for the purposes of testing our interpreter. Here are some such programs. You should, of course, first determine their value by hand (by writing the continuations in the form of \texttt{lambda} procedures, substituting, and evaluating).
First, a few old favorites, just to make sure the easy cases work correctly:

\[
\{\text{bindcc } k 3\} \\
\{\text{bindcc } k \{k 3\}\} \\
\{\text{bindcc } k \{+ 1 \{k 3\}\}\} \\
\{+ 1 \{\text{bindcc } k \{+ 1 \{k 3\}\}\}\} \\
\]

And now for three classic examples from the continuations lore (the fourth is just an extension of the third):

\[
\{\{\text{bindcc } k \{k \{k 3\}\}\} \{\text{fun } \{\text{dummy} \} \{k \{k 3\}\}\}\}\} \\
1729 \\
\{\text{bindcc } k \{k \{k 3\}\}\} \\
\{\{\text{bindcc } k k\} \{\text{fun } \{x\} \{x\}\}\} \{k 3\}\} \\
\{\{\{\text{bindcc } k k\} \{\text{fun } \{x\} \{x\}\}\} \{\text{fun } \{x\} \{x\}\}\} 3\} \\
\]

The answer in each case is fairly obvious, but you would be cheating yourself if you didn’t hand-evaluate each of these first. This is painful, but there’s no royal road to understanding!
(define-type CFAE-Value
    [numV (n number?)]
    [closureV (p procedure?)])

;; interp : CFAE SubCache receiver → doesn’t return
(define (interp expr sc k)
    (type-case CFAE expr
        [num (n) (k (numV n))]
        [add (l r) (interp l sc
            (lambda (lv)
                (interp r sc
                    (lambda (rv)
                        (k (num + lv rv))))))]
        [if0 (test pass fail) (interp test sc
            (lambda (tv)
                (if (num-zero? tv)
                    (interp pass sc k)
                    (interp fail sc k)))]
        [id (v) (k (lookup v sc))]
        [fun (param body) (k (closureV (lambda (arg-val dyn-k)
            (interp body (aSub param arg-val sc) dyn-k))))]
        [app (fun-expr arg-expr) (interp fun-expr sc
            (lambda (fun-val)
                (interp arg-expr sc
                    (lambda (arg-val)
                        ((closureV-p fun-val arg-val k))))))])

Figure 20.1: Making Continuations Explicit
\begin{quote}
\texttt{(define-type KCFAE-Value}
\begin{quote}
\texttt{[numV (n number?)]
\texttt{[closureV (p procedure?)]
\texttt{[contV (c procedure?)])}
\end{quote}
\end{quote}

\texttt{;; interp : KCFAE Env receiver \rightarrow doesn’t return
\texttt{(define (interp expr sc k)}
\begin{quote}
\texttt{(type-case KCFAE expr}
\begin{quote}
\texttt{[num (n) (k (numV n))]
\texttt{[add (l r) (interp l sc}
\begin{quote}
\texttt{(lambda (lv)}
\begin{quote}
\texttt{(interp r sc}
\begin{quote}
\texttt{(lambda (rv)}
\begin{quote}
\texttt{(interp k (num + lv rv))))))]
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\texttt{[if0 (test pass fail)
\begin{quote}
\texttt{(interp test sc}
\begin{quote}
\texttt{(lambda (tv)}
\begin{quote}
\texttt{(if (num-zero? tv)
\begin{quote}
\texttt{(interp pass sc k)}
\begin{quote}
\texttt{(interp fail sc k))}})
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\texttt{[id (v) (k (lookup v sc))]
\texttt{[fun (param body)
\begin{quote}
\texttt{(k (closureV (lambda (arg-val dyn-k)
\begin{quote}
\texttt{(interp body (aSub param arg-val sc) dyn-k)))])}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\texttt{[app (fun-exp arg-exp)
\begin{quote}
\texttt{(interp fun-exp expr sc}
\begin{quote}
\texttt{(lambda (fun-val)}
\begin{quote}
\texttt{(interp arg-exp expr sc}
\begin{quote}
\texttt{(lambda (arg-val)}
\begin{quote}
\texttt{(type-case KCFAE-Value fun-val}
\begin{quote}
\texttt{[closureV (c) (c arg-val k)]}
\texttt{[contV (c) (c arg-val)]}
\texttt{[else (error "not an applicable value")]))])}})
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\texttt{[bindcc (cont-var body)
\begin{quote}
\texttt{(interp body}
\begin{quote}
\texttt{(aSub cont-var}
\begin{quote}
\texttt{(contV (lambda (val)}
\begin{quote}
\texttt{(k val)))
\end{quote}
\end{quote}
\end{quote}
\begin{quote}
\texttt{sc)
\begin{quote}
\texttt{(k))}})
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
\end{quote}
}\end{quote}

Figure 20.2: Adding Continuations as Language Constructs