Implementing Continuations

sk and dbtucker

2002-10-18

1 Changing Representations

Now that we’ve seen how continuations work, let’s study how to implement them in an interpreter. For this lecture onward, please switch back down to the cs173 language level in DrScheme.

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

This change is pretty easy. First, we’ll modify the datatype of values. We’ll still use a datatype, so we can use a cases statement to distinguish between numbers and procedures:

(define-datatype CFA-value CFA-value? [numV (n number?)] [closureV (c procedure?)])

Only two rules in the interpreter need to change: that which creates values of this representation, and that which uses those values. These are the fun and app cases, respectively.

The new fun case simply creates an actual Scheme procedure. What is the body of that procedure? On invocation, the closure representation invokes the interpreter. Then that’s exactly what this procedure will have to do as well:

[fun (param body) (closureV (lambda (arg-val) (interp body (aSub param arg-val env))))]

The application case is also similarly brief:

[app (fun-expr arg-expr) (local ((define fun-val (interp fun-expr env)) [define arg-val (interp arg-expr env)] (cases CFA-value fun-val [closureV (c) (c arg-val)] [else (error 'interp "can only apply functions")])))]

And that’s it. Note that this change is so easy only because functions in the interpreted language have two properties: they are eager, and they obey static scope. That’s why Scheme’s lambda captures them so well. (As we’ve discussed before, this is our the usual benefit of using meta interpreters that match the interpreted language.)

2 Representing Continuations

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.

¹The purists amongst you might argue that we’re switching to a more meta interpreter, but this is okay for two reasons: (1) by now, you 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.
We’ll assume that the interpreter takes an extra argument \( k \), which we’ll call the 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: We never want to use an invocation of \( \text{interp} \) as a sub-expression of some bigger expression. Instead, we want \( \text{interp} \) to communicate its answer by passing it to the given \( k \). We’ll see the full motivation for this transformation soon, but for now, think of the interpreter as executing remotely over the Web. That is, each time we invoke \( \text{interp} \), 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.

This is pretty easy to explain directly through code. 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.

\[
\text{(define (interp expr env k)}
\text{ (cases CFAE expr)}
\text{ [num (n) (k (numV n))]}\]
\[\cdots)\]

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

\[
\begin{align*}
\text{[id (v) (k (lookup v env))]} \\
\text{[fun (param body)} \\
\text{ (k (closureV (lambda (arg-val)} \\
\text{ (interp body (aSub param arg-val env))))))]
\end{align*}
\]

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

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

The naïve solution might be to transform it as follows:

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

but do we have a value immediately handy to pass off to \( k \)? We will after interpreting the entire expression, but we don’t just yet. Recall that we can’t invoke \( \text{interp} \) in the midst of some larger computation, because it’ll be lost by the Web API. 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 naïve version, what’s the first thing the interpreter needs to do? It must evaluate the left sub-expression.\(^2\) So we write that first, and move all the remaining computation into the receiver of that invocation of the interpreter:

\[
\text{[add (l r) (interp l env}}
\text{ (lambda (lv) ;; lv for “value of l”)}
\text{ (k (numV+ lv (interp r env)))))]}\]

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 \( \text{interp} \) 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:

\[
\text{[add (l r) (interp l env}}
\text{ (lambda (lv) ;; lv for “value of l”)}
\text{ (interp r env}}
\text{ (lambda (rv) ;; “value of r”)}
\text{ (k (numV+ lv rv)))))]}\]

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 \( \text{interp} \) is no longer in a sub-expression—it’s always the first thing that happens in a receiver. What about the invocation of \( \text{numV+} \), though? Do we have to transform it the same way?

\(^2\)Notice that once again, we’ve been forced to choose an order of evaluation, just as we had to do to implement state.
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 `numV+` is just a wrapper around addition).³ 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:

```
[if0 (test then else)
  (interp test env
    (lambda (tv)
      (if (numV-zero? tv)
        (interp then env
          (interp else env
            · · ·)))))]
```

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. That is, the last two invocations of the interpreter are not, in fact, in a sub-expression position.

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`:

```
[if0 (test then else)
  (interp test env
    (lambda (tv)
      (if (numV-zero? tv)
        (interp then env k)
        (interp else env 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:

```
[app (fun-expr arg-expr)
  (interp fun-expr env
    (lambda (fun-val)
      (interp arg-expr env
        (lambda (arg-val)
          (cases CFA-value fun-val
            [case closureV (c)
              [c]]
            [else (error "interp "can only apply functions")]])))))
```

All we have to determine is what to write in place of the box.

What was in place of the box was `(c 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 ...
```

³Using our Web analogy, the question is which procedures are evaluated over the Web, and which ones locally. Ones that use the Web must be transformed and be given receivers to stash on the server, ones that don’t can remain unmolested.
Oops! DrScheme highlights the following boxed expression:

```
(fun (param body)
  (k (closureV (
      (lambda (arg-val)
        (interp body (aSub param arg-val env)))))
))
```

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

```
(fun (param body)
  (k (closureV (lambda (arg-val dyn-k)
        (interp body (aSub param arg-val env) dyn-k)))
))
```

(What happens if we instead use \(k\) instead of \(dyn-k\) in the invocation of the interpreter? Try it and find out!) Correspondingly, application becomes

```
(app (fun-expr arg-expr)
  (interp fun-expr env
    (lambda (fun-val)
      (interp arg-expr env
        (lambda (arg-val)
          (cases CFA-value fun-val
            (closureV (c)
              (c arg-val k))
            (else
              (error "interp "can only apply functions")))))))))
```

Incidentally, what’s the type of the interpreter? Obviously it now has one extra argument, but you can figure that out. 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. It is obsessed with not returning, which corresponds to our Web intuition: if the interpreter was invoked over the Web, the caller disappears by the time the interpreter is done, so there is nothing to return to!

## 3 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 basically the same thing as the continuation procedures we’ve been writing so far. They appear to differ in two ways, but:

- They capture what’s left to be done in the interpreter, not in the user’s program. But because the interpreter happens to close over the expressions of the user’s program, recursive invocations of the interpreter continue with the execution of the user’s program. Therefore, the receivers effectively do capture the continuations of the user’s program.

- They are regular Scheme procedures, not \(\text{lambda}\) procedures. We have, however, 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.

First, we’ll need to determine how we will extend the language. There are many ways to add continuations. The most spartan approach is the one traditional Scheme takes. It adds just one primitive, \(\text{call/cc}\), and treats the resulting continuation as procedures, so that procedure application does double duty. DrScheme slightly enriches the language by also providing \(\text{let/cc}\), but continues to overload procedure application. The functional language SML uses \(\text{callcc}\) to
capture continuations (so programmers must use the SML equivalent of \texttt{lambda} to name the continuations), but adds a \texttt{throw} construct to invoke continuations, so procedure application invokes only procedures, and \texttt{throw} invokes only continuations, making life easier for the type checker. We’re not aware of any language with both a binding construct like \texttt{letcc} and a separate \texttt{throw}-like construct for continuation invocation.

In a way, the traditional Scheme approach is truly insidious. Normally, procedural primitives such as + are extremely simple, often implementable directly in terms of a small number of standard machine code instructions. In that sense, continuations are extremely insidious: they masquerade as a procedural primitive, but they add immense power to the language and significantly change its semantics. This is arguably a bad design decision, because it fails to give the student of the language enough of a signpost of impending dangerous bends.

For this lecture, we’ll therefore take the DrScheme approach instead: add a binding construct but overload procedural application. (The latter decision is extremely easy to reverse; in a handful of nearly trivial lines you could add \texttt{throw} instead.) So, here’s our revised language datatype (the “K”, naturally, stands for continuations):

\begin{verbatim}
(define-datatype KCFAE KCFAE?
  [num (n number?)]
  [add (lhs KCFAE?) (rhs KCFAE?)]
  [if0 (test KCFAE?) (then KCFAE?) (else KCFAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body KCFAE?)]
  [app (fun-expr KCFAE?) (arg-expr KCFAE?)]
  [bindcc (cont-var symbol?) (body KCFAE?)])
\end{verbatim}

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 \texttt{contV}.

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

\[
\text{bindcc (cont-var body)} \rightarrow (\text{interp body} \ (aSub \ cont-var \ (contV \ ...) \ env) \ k)
\]

The receiver of the body is the same as the receiver of the \texttt{bindcc} expression: remember that if the continuation never gets used, evaluation must proceed normally.

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 \textit{creation}, not \textit{use}, of the continuation:

\[
\text{bindcc (cont-var body)} \rightarrow (\text{interp body} \ (aSub \ cont-var \ (contV \ (\lambda (val) \ (k \ val)) \ env) \ k))
\]

(Note again the reliance on Scheme’s static scope.) This makes the modification to the application clause very easy:

\begin{verbatim}
(cases KCFAE-value fun-val
  [closureV (c) (c arg-val k)]
  [contV (c) (c arg-val)]
\end{verbatim}
(error 'interp "can only apply functions")

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. In practice, it’s perfectly okay to use the identity function, which will result in the invocation of the interpreter resulting in a value for immediate use.

And that’s it! In these few lines, we have captured the essence of the meaning of continuations. 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 \( \texttt{let/cc} \) 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.

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

4 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 continuations do use state for auxiliary data structures, it’s possible to write some fairly complicated continuation programs without state for the purposes of testing our interpreter. Here are some programs whose value you should try to determine by hand before entering in the interpreter.

First, a few old favorites, just to make sure the easy cases work correctly:

\[
\begin{align*}
&\text{(bindcc } k \ 3) \\
&\text{(bindcc } k \ \{k \ 3\}) \\
&\text{(bindcc } k \ \{+ \ 1 \ \{k \ 3\}\}) \\
&\text{(+ \ 1 \ \text{(bindcc } k \ \{+ \ 1 \ \{k \ 3\}\}))}
\end{align*}
\]

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

\[
\begin{align*}
&\text{((bindcc } k \ \\
&\quad \{k \ \text{fun } \{\text{dummy}\} \\
&\quad \quad \quad \quad 3\}) \} \\
&\text{1729})
\end{align*}
\]

\[
\begin{align*}
&\text{(bindcc } k \ \\
&\quad \{k \ \\
&\quad \quad \{k \ 3\}\})
\end{align*}
\]

\[
\begin{align*}
&\text{({{bindcc } k \ k} \\
&\quad \{\text{fun } \{x\} \ x\})} \\
&\text{3}
\end{align*}
\]

\[
\begin{align*}
&\text{({{bindcc } k \ k} \\
&\quad \{\text{fun } \{x\} \ x\}) \\
&\quad \{\text{fun } \{x\} \ x\}) \\
&\text{3}
\end{align*}
\]

The answer in each case is fairly obvious, but you would be cheating yourself if you didn’t write down each of the continuations in its evaluation to step to your answer. Then walk the interpreter and make sure it uses the same continuations. This is painful, but there’s no royal road to understanding!