1 Introduction

Suppose we had a program of the form

\[
(\text{define} \ (\text{count-up-or-down}) \n
(\text{if} \ (\text{prompt-read} \ "\text{Count delay?}\ ")) \n
(\text{count-delay}) \n
(\text{countdown} \ (\text{prompt-read} \ \ldots)))\)\]

(\text{begin} \n
(\text{count-up-or-down}) \n
(\text{output} \ "\text{Done counting!}\"))\]

Now suppose we want to use the Web implementations of \text{prompt-read} and \text{output}. That means, in turn, we need to use the Web implementations of \text{countdown} and \text{count-delay}. We also need to create a Web version of \text{count-up-or-down}, which consumes an extra action argument, so we remember to eventually output the message that we’re done counting.

When we convert \text{count-up-or-down}, we must propagate its action along to whomever it calls. If the user asks to count the delay, we’re fine: just invoke \text{count-delay/web}. What if she asks to countdown instead? We don’t have a \text{countdown/web} that consumes an extra action argument. That means we’ll end up throwing away the action supplied to \text{count-up-or-down}, which in turn means when \text{countdown} is done, we’ll never print our completion message.

You might say, this is only a problem if the user asks to count down; otherwise we don’t need to worry. But how do we know what the user will choose? Even if we know a lot about this user’s preferences, we can’t in general know whether an expression will always return true instead of false. This is equivalent to something called the \text{Halting Problem}, which I assume you have seen before.

Why did we get into this situation? It’s because we did the conversion on a “by need” basis. In general, however, we won’t know for sure that a conversion isn’t necessary (as illustrated above). To be safe—in particular, to \text{preserve the meaning of our program}—we must always convert every procedure to its Web version. That’s what we will study below.
2 A Little Terminology and Background

Okay, I admit it: there’s no such thing as the Web transformation. At least, if you ask other professors who might teach this material, they wouldn’t have any idea what you’re talking about. But most of them will immediately recognize the name continuation-passing style (CPS).

CPS is one of the most venerable ideas in programming languages. It has formed the basis of most compilers for advanced languages like ML and Scheme; in fact, as some researchers pointed out about a decade ago, it underlies many compilers for languages like C and Fortran, even! In short, this is pretty Deep Stuff. We could teach a whole course dedicated to this one topic, and have plenty left over at semester’s end. Indeed, we will see quite a bit more of CPS during the course of this semester.

One of the remarkable, recent research results is that CPS is also at the heart of Web programming. The style of chaining (jumping from one (logical) script to the next) that CGI programmers use, and CGI programming books recommend, is essentially a half-formal, sometimes-buggy, hand-generated version of CPS. In a course like this, you would normally see CPS first, and Web programming maybe never at all. What I’ve tried to do, instead, is have you build up an intuition for Web programming first, and let in on the fact that this is just CPS later.

One little bit of terminology you should know is that the “action” field is formally called a continuation: it represents what’s left to be done, i.e., how to continue the computation. Hence the expansion of CPS: instead of “doing stuff”, you’re always passing continuations around to someone else, expecting them to “do it” and just give the continuation the resulting value. If you prefer, you could call it bucket-brigade programming.

3 The Continuation-Passing-Style Transformation

CPS results from a transformation. That is, a function consumes a program, and generates another program that is equivalent in behavior but (usually) different in form from the original. In particular, it consumes and produces programs in the same language. We therefore need a language powerful enough to represent the changes wrought by CPS. Both Scheme and Rip fit the bill just fine (whereas converting Java programs to CPS requires quite a bit more work). We’ll use Rip, since that’s the language you’ve been implementing all semester.

On to the transformation. We’ll define CPS in two stages. First, we handle syntactic values. Every expression reduces to a value (unless it represents an infinite loop!), but most of them require one or more steps of computation to do so. To us, computation means, roughly, “stuff we need to keep track of”. That means we need to create continuations. When can we avoid doing this? When the expression is syntactically a value, that is, it requires zero computation steps to reduce to a value. In Rip, the syntactic values are
• numbers
• variables
• procedures

(Yes, procedures: they perform computation when you apply them to a value, but by themselves they’re sedentary.)

Converting the first two kinds of syntactic values to CPS is trivial: leave them alone. The third is just a little more tricky. We want to convert

{proc {x} body}

into

{proc {x}
  {proc {dyn-k}
    body'}}

where dyn-k is the dynamic continuation: the body of work waiting to be done when the application of this procedure reduces to a value. We need to feed dyn-k the value that body reduces to. It’s not immediately clear how we can do this; who knows how complex body might be? So we’ll assume that body’ is just a converted version of body that feeds its result to dyn-k.

Why did we decide that the result of CPSing a procedure should be of the form

{proc {x}
  {proc {dyn-k} . . .}}

rather than

{proc {dyn-k}
  {proc {x} . . .}}

? There’s no good reason; it’s just a convention. We just need to make sure that whatever we write here, we must consistently define the CPS of a procedure application, so we correctly line up values with values and continuations with continuations. If Rip had procedures that accepted two arguments, we could have passed in both the argument value and the continuation at the same time.

But these are not very interesting questions . . .

Here’s the CPS converter for syntactic values:

\[
\text{(define (cps/value expr)}
\]
\[
\text{(cond}
\]
\[
\text{[ (numE? expr) expr]}\]
\[
\text{[ (varE? expr) expr]}
\]
\[
\text{[ (procE? expr)}
\]
\[
\text{ (make-procE (procE-arg-name expr)}
\]
\[
\text{ (cps/expr (procE-body expr)))]])\]

3
We clearly need to define \( cps/expr \) next. We’ve seen how to represent syntactic values: two are pretty simple, the third is a little more complex, but makes sense with some thought. Now we need to consider the representation of an expression \( E \). Almost every \( E \) is going to be in the midst of some larger expression, \( E' \). Normally, \( E' \) would await the evaluation of \( E \), and use its value to be done evaluating. In the Web/CPS world, however, nothing waits for values, in case the program performs output and then halts, forgetting about waiting expressions. Therefore, expressions like \( E' \) therefore turn into continuations to which we must pass the value of \( E \) when it becomes available.

That last bit is worth repeating: expressions ... turn into continuations. That is, an expression becomes a procedure of one argument that eventually supplies its value to that argument. Translated into code, this becomes:

\[
\begin{align*}
&\textbf{(define } (cps/expr expr) \\
&\textbf{(let } ([k (gensym 'k)]) \\
&\textbf{(make-procE k} \\
&\textbf{(cps/expr/k expr (make-varE k))})))
\end{align*}
\]

(Puzzle: Why the \( gensym \)? What goes wrong if we don’t use it? Can you generate a DrScheme transcript that demonstrates the problem?)

Okay, now we’re left with the hard part, which is \( cps/expr/k \). At least we know that whenever we get a value, we must pass it to the second argument of this procedure. Let’s consider its definition’s clauses in order.

This is just getting the easy bits out of the way:

\[
\begin{align*}
&\textbf{(define } (cps/expr/k expr kont) \\
&\textbf{(cond} \\
&\textbf{[(syntactic-value? expr) \\
&\textbf{(make-appE kont (cps/value expr))}]}
\end{align*}
\]

If we have an expression of the form

\[
\{ + \{ +34 \} \{ +12 \} \}
\]

we’d like to convert it into

\[
\{ \text{proc } \{k}\} \\
\{ \{ +34 \} \} \\
\{ \text{proc } \{lhsV}\} \\
\{ \{ +12 \} \} \\
\{ \text{proc } \{rhsV}\} \\
\{ \{ k \{ + lhsV rhsV\} \} \}
\]

That is, we first reduce each argument to the outer addition down to a value \((lhsV\) and \(rhsV\), respectively). We then add these two values to get the answer. We feed that answer to the continuation waiting for it, namely the argument to the outermost \( \text{proc} \). Because this outermost procedure is generated by \( cps/expr \), we only need to generate the innards:

\[
\textbf{[(addE? expr)}
\]
(let ([lhsV (gensym 'lhsV)]
    [rhsV (gensym 'rhsV)])
  (cps:expr/k (addE-lhs expr)
    (make-procE lhsV
      (cps:expr/k (addE-rhs expr)
        (make-procE rhsV
          (make-appE kont
            (make-addE
              (make-varE lhsV
                (make-varE rhsV))))))))
)

That leaves one last clause, procedure application. This is almost the same as the conversion of addition, with one small but key difference:

[(appE? expr)
  (let ([procV (gensym 'procV)]
      [argV (gensym 'argV)])
    (cps:expr/k (appE-proc expr)
      (make-procE procV
        (cps:expr/k (appE-arg expr)
          (make-procE argV
            (make-appE
              (make-appE (make-varE procV)
                (make-varE argV))
              kont))))))]

You might wonder, “+ is a procedure too, so why would these two be different at all?” (I hope you wonder about this.) In reality, the correct, safe CPS version of both would look the same (with a small adjustment for the number of arguments). If we had any doubt at all about how + behaved, we’d want it converted to CPS, too. Then we’d pass it a continuation, and let it invoke whatever it wanted. We’re assuming here, instead, that + is well-behaved, never performs i/o (on the Web, say), and returns almost immediately with an answer.

For an arbitrary, user-defined procedure, we have no such guarantees. Therefore, we adopt the more conservative transformation. This passes the dynamic continuation on to the procedure (the “second” argument to the transformed representation of procedures), letting it perform whatever operations it wants without affecting the correctness of the result.

All that leaves is a procedure to set off the process:

(define (cps expr)
  (if (syntactic-value? expr)
      (cps/value expr)
    (let ([x (gensym 'x)])
      (make-appE (cps:expr x)
        (make-procE x (make-varE x))))))
This provides the identity function as the initial continuation. Read this as saying “there’s nothing left to be done”. The initial continuation is fed the final value of the computation. In an interactive environment, such as DrScheme, the initial continuation would be a bit more complex: it would accept a value, print it, then print a new prompt, read an expression, parse it, and run it in the same initial continuation.

You might wonder whether you need to create a new interpreter to run Rip programs converted to CPS. Remember, however, that CPS is just a stylistic restriction on Rip. Therefore, every Rip program in CPS is just that: a Rip program. To test our handiwork, therefore, we can define

\[
\begin{align*}
&\text{(define (interp/cps expr env)} \\
&\quad (\text{interp (cps expr)} env))
\end{align*}
\]

and map interp/cps over our test suite to ensure our CPS conversion routines preserve the meaning of the program.