Continuation Passing Style

Mattox Beckman

Introduction

A continuation takes the idea of a function's return value and generalizes it. Normally a function returns a value to the expression that called the function in the first place. But what if a function could return *somewhere else* instead? This idea of "somewhere else" is called a *continuation*.

Basic Continuations

Direct Style

The functions you have been writing until now have been in a form known as *direct style*. The functions take an argument, and then return a result to the calling expression.

Consider the simple example in figure 1.

```
dec a = a - 1
double a = a * 2
inc a = a + 1
report a = a
```

Figure 1: Direct Style Example

We can run it like this:

```
Main> double (dec 3)
4
```

How does this happen? The first thing to execute is the call to `dec`. Function `dec` consumes the 3, and when it is finished, returns its result to the surrounding expression. We are left with the expression `double 2`. Consider for a moment the role of `double` in this expression. The `double` • part of the expression can be considered an expression with a hole where the • is. When `dec` returns, that hole is filled with the result, and the expression continues. For this reason, `double` is called the *continuation* of `dec`.

Consider another example:

```
Main> dec (double (inc 20))
41
```

The first function to be called is `inc`. When it is done, it will return its result to the surrounding expression, which we can represent as `dec (double •)`

— its continuation. Function `double` consumes the result and returns 42 to its own continuation `dec •`.

One way to think of a continuation is that it's a way of identifying the *rest of the program*.

Continuation Passing Style

You will notice that the expressions we called continuations were like normal expressions, but each had a hole which we could fill with a value later. A function is similar. So similar, in fact, that we can implement continuations by using functions. Consider the program in figure 2. Each of these functions takes an additional argument `k`, which represents "what comes next" in the program.

```
cdec a k = k (a - 1)
cdouble a k = k (a * 2)
cinc a k = k (a + 1)
report a = a
```

Figure 2: Continuation Passing Style Example

First, a very simple example.

```
Main> cdec 3 report
2
```

Here, `cdec` took an argument 3 and a continuation `report`. When it had finished performing its computation with 3, it passed the result to `report` rather than returning the result to the calling expression. This is one of the properties of functions written in CPS: they do not return[1]. Instead, they call their continuation directly.

Another way to look at this is to say that a continuation is a generalization of a `return` keyword. Instead of returning to the expression which called it, a function can pass its result to an arbitrary location. (Imagine what our code would look like if we used the name `return` (which is not a keyword in Haskell) instead of `k`.)

[1] Truth in advertising: in Haskell, the `report` function actually does return the final result. But you will notice that it is a tail call, so the computation is over by the time that happens. In a language that supports real continuations, a "final" continuation such as `report` would not return at all.

We can run the first example in (CPS) like this:

```
Main> cdec 3 (\ r -> cdouble r report)
4
```

What happens is that `cdec` takes 3 as an argument and `( r -> cdouble r report)` as a continuation. When `cdec` is done, it passes its result to the continuation instead of returning it. The resulting expression is `cdouble 2 report`. Now a similar thing happens. First `double` consumes the 2 and, when done, passes the result to its own continuation `report`.

Our second example looks like this:

```
Main> cinc 20 (\ r1 -> cdouble r1 (\ r2 -> dec r2 report))
41
```

It is only slightly more involved than the first example.

Note that we can rewrite the example this way:

```
Main> cinc 20 (\ r ->
  cdouble r (\ r ->
  cdec r
  report))
41
```

Perhaps this will remind you of an imperative language, or even an accumulator based assembly language.

Let's look at a more complicated example.

```
cinc a k = k (a+1)
cdec a k = k (a-1)
cadd a b k = k (a+b)
report a = a
```

Figure 3: Nesting Continuations

Think for a moment how you would convert the following direct-style expression to the code in figure 3.

```
Main> add (inc 4) (dec 5)
- : int = 9
```

The first thing to be executed is `inc`, so we would call `cinc` first, and save its result in the continuation. Next, we would want to call `cdec`, and again save its result. Finally, with the two saved results, we can call `cadd`, and pass the final answer to `report`.

```
Main> cinc 4 (\ a ->
  cdec 5 (\ b ->
  cadd a b report))
9
- : unit = ()
```

Stare at that for a while and make sure you understand how it works. We are taking advantage of the fact that one continuation is nested inside of another, and that the inner continuation (the `b -> ...` one) has access to the scope of the outer one.

But now consider this code:

```
Main> cdec 5 (\ b ->
  cinc 4 (\ a ->
  cadd a b report))
```

```
9
- : unit = ()
```

It is different from the previous example in only one way: the order of operations has changed. In the direct style example, the compiler has the option (in some langauges, such as C) of evaluating *dec* and `inc` in any order. But, if we use CPS, then there is only one order of operation for the expression. What CPS has done for us is it has *exposed the program's flow of control*, and allowed us to constrain it.

In review, this is what we have so far:

- A function written in CPS doesn't return. Instead it passes its result to another function.
- A continuation represents "the rest of the computation".
- The order of operations in CPS code is made explicit, and constrained.
- Calls to continuations are always tail-calls.

Continuations and Recursion

Continuation passing style can do some very interesting things when we mix it with recursion.

Consider the program in figure 4. It simply multiplies a list of integers together.

Figure 4: Simple Recursion

```
multlist [] = 1
multlist (x:xs) = x * (multlist xs)
```

If we were to run it with the input `[2, 3, 4]` we would get the execution trace in figure 5.

Figure 5: Simple Recursion Sample Run

```
multlist [2, 3, 4]
2 * (multlist [3, 4])
2 * (3 * (multlist [4]))
2 * (3 * (4 * (multlist [])))
2 * (3 * (4 * 1))
2 * (3 * 4)
2 * 12
24
```

As the computation progresses, we stack up recursive calls until we reach the base case, and then return from all the recursive calls to compute the result.

Consider what's happening in line 2 of the sample run: `2 *` • is the continuation, which will receive the result of `(multlist [3, 4])`.

Accumulator Recursion

Another style of recursion avoids the use of the stack by accumulating the value in a separate parameter. The code is in figure 6.

If we were to run it with the input `[2, 3, 4]` and 1 we would get

```
amlutlist []       = acc
amlutlist (x::xs) = amultlist xs (x * acc)
```

Figure 6: Accumulator Recursion

```
amultlist [2, 3, 4] 1
amultlist [3, 4] (* 2 1)
amultlist [3, 4] 2
amultlist [4] (* 3 2)
amultlist [4] 6
amultlist [] (* 4 6)
amultlist [] 24
24
```

Figure 7: Sample Run of figure 6

Notice this time that we only need to keep track of a single call to `multlist` at any one time... the result of one call is simply the value of the next call. This is known as `tail-recursion`, and is useful because we do not need to keep track of where in the main expression the result needs to be placed. The result is not going to be handed back to anyone to be processed further, as it was in figure 4. It is simply returned.

When written in this form, the compiler will eliminate the call-stack, which will speed things up, and reduce the amount of memory we need on the stack.

Notice how the computations are performed. In the simple recursion example, the recursions happened first, and the computation occurred as we returned from the recursions. In this examples, the computation occurs first, and are passed into the recursive call.

Continuation Passing

Figure 6 accumulates a value to be given to the recursive call, but that's not the only kind of accumulation we can make. Instead of accumulating values, we can also accumulate *computations*. Consider figure 8, which is the same code written in CPS.

```
kmultlist k []       = k 1
kmultlist k (x::xs) = kmultlist xs (\ r -> k (r * x))
```

Figure 8: Continuation Passing `multlist`

Again, the main idea behind CPS is that *functions never return*, and therefore you need to specify what should happen next when a result is computed.

This is done by giving each function an extra argument, called the *continuation*. This argument usually comes last, and is often called `k`. When the funtion is finished computing its result, it will pass that result into `k` instead of returning.

If figure 8 we can see how the continuations are used. If `kmultlist` is called with the empty list, it is as the base case, which is defined to be 1. So, `kmultlist` passes 1 to `k`. The recursive case is more complex, and may seem unusual at first. Suppose that `kmultlist` is called with some non-empty list `xx`—refer to this as the initial call. In order to compute the result, `kmultlist` needs to make a recursive call on the tail of `xx`. Since this recursive call will not return, we need to give it a continuation to tell it what to do with that result. This continuation should save the result of the recursive call, multiply it by the head of `xx`, and then pass that new result to the continuation given to the initial call. All this is done in the last line. The variable `r` saves the result of the recursive call to `kmultlist`, the continuation then multiplies `r` by `x`, and then passes the result to the initial continuation `k`.

An execution trace is given in figure 9.

```
kmultlist [2,3,4] report
kmultlist [3, 4] (λ v1 -> report (v1 * 2))
kmultlist [4] (λ v2 -> (λ v1 -> report (v1 * 2)) (v2 * 3))
kmultlist [] (λ v3 -> (λ v2 -> (λ v1 -> report (v1 * 2)) (v2 * 3)) (v3 * 4))
(λ v3 -> (λ v2 -> (λ v1 -> report (v1 * 2)) (v2 * 3)) (v3 * 4)) 1
(λ v2 -> (λ v1 -> report (v1 * 2)) (v2 * 3)) (1 * 4)
(λ v2 -> (λ v1 -> report (v1 * 2)) (v2 * 3)) 4
(λ v1 -> report (v1 * 2)) (4 * 3)
(λ v1 -> report (v1 * 2)) 12
report (12 * 2))
report 24)
24
```

Figure 9: Sample Run of Continuation Passing `multlist`

As you can see, at each stage of the recursion we take the old continuation and build a new, larger continuation out of it. Consider line two as an example. The function `kmultlist` is supposed to compute the product of `[2, 3, 4]`, and pass the result to `report`. To do that, it's going to call itself recursively, on the list `[3, 4]`. The result (12) will be placed into v_1, and then the function will multiply it by the current element of the list `[2]`. The current call to `kmultlist` is now ready with its result (24), so it is given to the continuation specified for this particular call (i.e., `report`).

In all of these coding styles, the problem is split into two parts. To multiply the elements of a list, you first multiply the elements of the rest (i.e., the tail) of the list, and when you're done doing that, you multiply the result to the first element of the list.

- In the simple recursion, the recursive call multiplies all the elements to-

gether and returns the result to the initial call.

- In the accumulator recursion, the multiplication occurs first, and is given to the recursive call.
- In the CPS version, the recursive call builds a function which, when called with the base case, will perform the computation.

Multiple Continuations

You can think of continuations as being a return address, made explicit. Because continuations can be stored in variables, there is no reason we can't keep more than one of them around. In figure 10 there is a new version of the `kmultlist` function, that can "bail out" of a computation. The front end to this function keeps the original continuation, and sends a copy to the auxiliary function. The copy (`kr`, the "result" continuation) is used as before, but the original (`k`, the "abort" continuation) is kept unchanged. The continuation `kr` represents the function's computation; call it, and the computation occurs. The `k` continuation, if called, will skip all that computation and (appear to!) exit out of all the levels of recursion, back to the beginning.

```
kmultlist xx k = aux xx k
  where aux [] kr = kr 1
        aux (0:xs) kr = k 0
        aux (x:xs) kr = aux xs (\v -> kr (v * x))
```

Figure 10: Aborting Continuation

Figure 11 shows a sample run where the abort continuation is called. Again, the `kr` continuation allows you to pass a result back to the previous recursive call's computation, and the `k` continuation allows you to pass a result back to the initial function call.

```
kmultlist [2, 3, 0] report
aux [2, 3, 0] report
aux [3, 0] (λ v1 -> report (v1 * 2))
aux [0]) (λ v2 -> (λ v1 -> report (v1 * 2)) (v2 * 3))
report 0) — here we've thrown out kr and called k instead
0
```

Figure 11: Sample Run of Continuation Passing `multlist`

For that matter, why stop at two continuations? This next version interprets negative numbers in a strange way: if it finds $-n$, it undoes n levels of recursion, and activates the computation at that point.

The following sample run may illustrate the operation of the new function. Be sure you understand how it works.

```
Main> kmultlist [2, 3, 4, 5, 6, 1] report
720
```

```
nth 0 (x:xs) = x
nth n (x:xs) = nth (n-1) xs

kmultlist xx k = aux xx [k]
  where aux [] klist = (head klist) 1
        aux (0:xs) _ = k 0
        aux (x:xs) klist | x < 0 = (nth (0-x) klist) 1
                         | otherwise = aux xs ( (\v -> (head klist) (v * x)) :: klist)
```

Figure 12: Aborting Continuation, Part 2

```
- : unit = ()
Main> kmultlist [2, 3, 4, 5, 6, 1, 0, 2, 4, 6] report
0
- : unit = ()
Main> kmultlist [2, 3, 4, 5, 6, -1, 0, 2, 4, 6] report
120
- : unit = ()
Main> kmultlist [2, 3, 4, 5, 6, -2, 0, 2, 4, 6] report
24
- : unit = ()
```

Continuations, then, are a means of time travel. A continuation allows us to save a point of the program and return to it any time we want.