Lambdas Are Simple, or How To CPS

Paulette Koronkevich

October 23, 2018

We've been talking about how to convert functions to *continuation passing style,* but what does that even mean? A *continuation* is a way to analyze the *control flow* of your program at any point. The way we represent a continuation in CSCI-C311 is with a *function.* We transform our programs to "push" the computation into the continuation, so that we can track its evaluation. Now, here's a quick introduction on how to CPS (convert into continuation passing style) programs.

1 Change the name

Change the name of the program from `function-name` to `function-name-cps`. Easy enough, right?

2 Add a `k` to the formal parameters

`(define function-name (λ (`x_1 `...` x_n`) ...))`
changes to
`(define function-name-cps (λ (`x_1 `...` x_n `k) ...))`
Again, easy enough, right? This `k` represents the continuation. Remember that it is a function.

3 Apply `k` to simple values

Simple values include (but are not limited to) numbers, lists, booleans, strings, symbols, and most importantly - **lambdas are simple**.

```
(define add2 (λ (x) (+ 2 x)))
```
changes to
```
(define add2-cps (λ (x k) (k (+ 2 x))))
```

```
(define if0 (λ (x) (if (zero?  x) 'zero 'not-zero)))
```
changes to
```
(define if0-cps (λ (x k) (if (zero?  x) (k 'zero) (k 'not-zero))))
```

Notice that `k` is applied to the values *returned* by the function. Applying `k` to the entire `if` statement might work, but `if` isn't a simple value - its a way to introduce a branch in our program. We are trying to track how the values change throughout the program. Applying the continuation to simple values invokes the current "stack" of computation onto the value, and it doesn't really make sense to apply the evaluation of the whole program onto an `if` statement; rather, it should be done on individual values.

```
(define cur-mult (λ (n) (λ (m) (* n m)))
```
changes to
```
(define cur-mult-cps (λ (n k) (k (λ (m k) (k (* n m))))))
```

Notice that `k` is added to the formal parameters of the function returned by `cur-mult`. This is because *all functions used and returned by the program must also be in continuation passing style.* `cur-mult` is a function that, when given a number, returns a function. Lambdas are simple, so `k` must be applied onto this simple value `cur-mult` returns. The function returned also takes a number, and then multiplies the two. This function must be CPS'd along with `cur-mult`. Thus, we add `k` to the formal parameters, and apply `k` to the simple value it returns.

4 Convert all "serious" calls to tail calls

Um, what? Let's examine this step closely. First of all, "serious" calls are (1) recursive function calls, (2) calls to other, previously defined functions and (3) calling a function passed as an argument. Let's identify all serious calls in the following function:

```
(define srs-calls
```

```
(lambda (f x)
   (if (zero?  x) (f x) (+ 1 (srs-calls f (sub1 x))))))
```

There's an application of some argument called `f` onto the argument `x`, and also a recursive call to `srs-calls`. Thus, the two "serious" calls are highlighted in red:

```
(define srs-calls
   (lambda (f x)
      (if (zero?  x) (f x) (+ 1 (srs-calls f (sub1 x))))))
```

Alright, so what is a "tail call"? Unfortunately, it doesn't have to do with any furry tailed friends. A *tail call* is the application of a function where no computation is "waiting" on it. Refering back to our example, `(f x)` has no computation waiting on it, whereas `(srs-calls f (sub1 x))` has the `+ 1` waiting on its result.

4.1 Conversion

Now, how do we do this conversion? First and foremost, recall that *all functions used and returned by the program must also be in continuation passing style.* In order to convert serious calls to tail calls, we defer any computation waiting on it to its continuation. Let's see some simple examples.

```
(define simple (λ (f x) (f x)))
```
Applying the first two steps of CPSing:
```
(define simple-cps (λ (f x k) (f x)))
```

`f` must be in continuation passing style, so it must have a continuation. Recall that the continuation is a function that tracks the control flow of the program. We can start by writing a lambda to start constructing the continuation (function)...

```
(define simple-cps (λ (f x k) (f x (λ (res) ...))))
```

I named the formal parameter of this function `res` to emphasize the point that this continuation associated with this serious call will take as input the *result* of calling `f` on `x`! Once we have this result, we can apply any compu-

tation waiting on it, like the `+ 1` that waits on the serious call `(srs-calls f (sub1 x))` in the previous example. In this case, no computation is waiting on the result of `f` applied to `x`. This result is simple, since we can assume that `f` will terminate with a simple value. And what do we do with simple values? We apply `k`!

```
(define simple-cps (λ (f x k) (f x (λ (res) (k res)))))
```

Finally, notice that `(λ (res) (k res))` is equivalent to just `k`. This is called *eta-reduction*. It's not a necessary step, but makes our code look a lot cleaner in most cases, and I will be applying this reduction to future examples. Finally, we have:

```
(define simple-cps (λ (f x k) (f x k)))
```

Alright, more examples! Let's go back to `srs-calls`, now with the first two steps of the transformation applied:

```
(define srs-calls-cps
   (lambda (f x k)
      (if (zero?  x) (f x) (+ 1 (srs-calls f (sub1 x))))))
```

`(f x)` is easy enough to convert to a tail call - it's already one!

```
(define srs-calls-cps
   (lambda (f x k)
      (if (zero?  x) (f x k) (+ 1 (srs-calls f (sub1 x))))))
```

`(srs-calls f (sub1 x))` is not a tail call, so we need to "push" the computation that is waiting on the result into the continuation.

```
(define srs-calls-cps
   (lambda (f x k)
      (if (zero?  x)
         (f x k)
         (srs-calls f (sub1 x) (λ (res) (+ 1 res))))))
```

Finally, don't forget what we do with simple values! Apply `k`.

```
(define srs-calls-cps
   (lambda (f x k)
      (if (zero?  x)
         (f x k)
         (srs-calls f (sub1 x) (λ (res) (k (+ 1 res)))))))
```

A good rule of thumb to check through your program and make sure there is no computation waiting on a serious call. If you ever see something like `(add1 (f x))` or `(cons (f x) '())`, you're not done or have done something wrong. Keep applying this technique, "pushing" this computation into the continuation of the serious call.

Also, functions like cons, append, plus, minus, etc are **not serious**. These are operations over simple values, and return simple values.

Another rule of thumb: **start with the most deeply nested serious call**. What this means is find with the serious call on which other computation or serious calls rely on, compute its result first, and compute the result of other serious calls within its continuation. Let's practice finding the most deeply nested serious calls.

```
(f (h 3) x)
```

See that the serious call `f` onto `(h 3)` and `x` relies on the result of the serious call `(h 3)`, thus `(h 3)` is what we would want to compute first.

```
(h 3 (λ (res) (f res x k)))
```

Another example:

```
(if (d 5 6) (f (g 4)) (f (g 10)))
```

It may look like the `(g 4)` and `(g 10)` serious calls are the most deeply nested, however, they won't even be computed unless `(d 5 6)` is evaluated because of the `if`! Thus, the serious calls `(f (g 4))` and `(f (g 10))` *rely on* the result of `(d 5 6)`!

```
(d 5 6 (λ (res_d) (if res_d
                      (g 4 (λ (res_g4) (f res_g4 k)))
                      (g 10 (λ (res_g10) (f res_g10 k))))))
```

See how we are thinking about the control flow of our programs while CPSing. This is not an accident! This is the whole point of having a continuation.

5 A big example

As the name implies, don't try to understand this function, I'm just trying to show how we can CPS any function using these steps.

```
(define nonsense
   (λ (p g)
      ((λ (f)
         (add1 (f (g 6) 2)))
      (λ (x y)
         (cond
            [(zero?  x) y]
            [(p y) (add1 (g x))]
            [else ((g g) (nonsense g p) (add1 x))])))))
```

Applying the first two steps:

```
(define nonsense-cps
   (λ (p g k)
      ((λ (f k)
         (add1 (f (g 6) 2)))
      (λ (x y k)
         (cond
            [(zero?  x) y]
            [(p y) (add1 (g x))]
            [else ((g g) (nonsense-cps g p) (add1 x))])))))
```

Notice that the body of this function is one giant application of `(λ (f) (add1 (f (g 6) 2)))` onto `(λ (x y) (cond ...))`. Recall that *all functions used and returned by the program must also be in continuation passing style*, so that is why these functions also get `k` in their formal parameters. Since this is one giant application, let's abstract and define `(λ (f) (add1 (f (g 6) 2)))` to be `f` and `(λ (x y) (cond ...))` to be `x`, just to make this a little easier to visualize. We then have:

```
(define nonsense-cps
   (λ (p g k)
      (f x)))
```

How do we CPS this? We pass `k` to the application of `f` onto `x`!

```
(define nonsense-cps
   (λ (p g k)
      (f x k)))
```

Replacing `f` and `x` with what we had defined them to be earlier:

```
(define nonsense-cps
   (λ (p g k)
      ((λ (f k)
         (add1 (f (g 6) 2)))
      (λ (x y k)
         (cond
            [(zero?  x) y]
            [(p y) (add1 (g x))]
            [else ((g g) (nonsense-cps g p) (add1 x))])) k)))
```

Let's again focus on `f`, or `(λ (f) (add1 (f (g 6) 2)))`:

```
(λ (f k) (add1 (f (g 6) 2)))
```

Here, `(g 6)` is the most deeply nested serious call. We compute this, and then pass the result to `f` along with 2. Once we get this result, we `add1` to it, and since it is simple, we apply `k`:

```
(λ (f k)
   (g 6 (λ (res_g)
      (f res_g 2
         (λ (res_f) (k (add1 res_f)))))))
```

Now that `f` is finished, let's turn our attention to `x`, or `(λ (x y) (cond ...))`.

```
(λ (x y k)
   (cond
      [(zero?  x) y]
      [(p y) (add1 (g x))]
      [else ((g g) (nonsense-cps g p) (add1 x))]))
```

Again, it's easy to think that `(g g)` or `(nonsense-cps g p)` is the most deeply nested call, but note that we won't even get to the `else` case unless the serious call `(p y)` is computed! Let's start by computing `(p y)`, and placing the whole `cond` expression into its continuation, with `(p y)` replaced with $\texttt{res}_p$.

```
(λ (x y k)
   (p y (λ (res_p)
      (cond
         [(zero?  x) y]
         [res_p (add1 (g x))]
         [else ((g g) (nonsense-cps g p) (add1 x))]))))
```

Great, now we can focus on each `cond` case individually. The first two cases are quite easy:

```
(λ (x y k)
   (p y (λ (res_p)
      (cond
         [(zero?  x) (k y)]
         [res_p (g x (λ (res_g) (k (add1 res_g))))]
         [else ((g g) (nonsense-cps g p) (add1 x))]))))
```

The application of `(g g)` waits for `(nonsense-cps g p)`, but `(g g)` is a serious call itself. We start with computing `(g g)`, then `(nonsense-cps g p)`, then finally applying the result of `(g g)` onto `(nonsense-cps g p)` and `(add1 x)`, passing `k` since this application is already a tail call.

```
(λ (x y k)
   (p y (λ (res_p)
      (cond
         [(zero?  x) (k y)]
         [res_p (g x (λ (res_g) (k (add1 res_g))))]
         [else (g g (λ (res_g)
                  (nonsense-cps g p (λ (res_n)
                     (res_g res_n (add1 x) k)))))])))))
```

Putting our CPS'd `f` and `x` together, we finally get:

```
(define nonsense-cps
   (λ (p g k)
      ((λ (f k)
         (g 6 (λ (res_g)
               (f res_g 2
                  (λ (res_f) (k (add1 res_f)))))))
      (λ (x y k)
         (p y (λ (res_p)
               (cond
                  [(zero?  x) (k y)]
                  [res_p (g x (λ (res_g) (k (add1 res_g))))]
                  [else (g g (λ (res_g)
                           (nonsense-cps g p (λ (res_n)
                              (res_g res_n (add1 x) k)))))])))) k)))
```