Lesson 11
CONTINUATION-PASSING STYLE

June 20, 2023

1. Pipelines
2. Continuation-Passing Style
3. CPS Translation
4. Control Flow

Last lecture, we explored more applications of **higher-order functions**. In particular, we talked about how we can take advantage of **currying** to create **staged** functions, which can do useful work by shifting around computations with respect to when curried arguments are taken in.

We then looked at more examples of HOFs in action, such as by generalizing `map` and `fold` to tree data structures.

Finally, we looked at using the `|>` operator to sequentialize our code for enhanced readability, as well as the **option monad**, which let us reduce the amount of boilerplate code we needed to write.

1 - Pipelines

We defined the |> operator before as

```
infix |>
fun x |> f = f x
```

This lets us produce code like

```
[1, 2, 3]
|> map Int.toString
|> foldr (fn (x, y) => x ^ "," ^ y) ""
```

This works pretty well!

What if we were doing something slightly different? Suppose we now have a list of strings, and we want to interpret them as integers and find the average.

```
["1", "2", "3"]
|> map Int.fromString
|> map Option.valOf
|> (fn L => (foldr op+ 0 L) div (List.length L))
```

We have to introduce a lambda, because we want to use the value of `L` twice. We need to know how much the denominator is.

This is kind of gross, and breaks our nice sequencing, though! Because of this `div` function, it's not clear what the control flow of our function is. Before, our steps were cleanly separated by pipes – now, we rely on evaluation order. Can we do better?

Recall the idea of **eta expansion**[1], which is that any valuable function expression `f` is extensionally equivalent to putting it into a lambda which explicit names its argument, like `fn x => f x`.

Let's use that to explicitly name each result.

```
["1", "2", "3"]
|> (fn L1 =>     map Int.fromString L1
|> (fn L2 =>     map Option.valOf L2
|> (fn L =>      foldr op+ 0 L
|> (fn total =>  List.length L
|> (fn len =>    total div len )))))
```

[1] See Lecture 10: Combinators and Staging if you need a refresher.

Actually, we usually prefer to indent it slightly differently:

```
["1", "2", "3"]        |> (fn L1 =>
map Int.fromString L1  |> (fn L2 =>
map Option.valOf L2    |> (fn L =>
foldr op+ 0 L          |> (fn total =>
List.length L          |> (fn len =>
total div len          )))))
```

Now, each expression, such as `map Int.fromString L1`, is placed on the same line as the lambda expression that it is being piped into, which explicitly names the result of the computation (in this case, `L2`).

Now this looks nice and sequential!

In case you're having trouble reading the previous code, this might be a little more clear:

```
["1", "2", "3"] |> (fn L1 =>
  map Int.fromString L1 |> (fn L2 =>
    map Option.valOf L2 |> (fn L =>
      foldr op+ 0 L |> (fn total =>
        List.length L |> (fn len => total div len)
      )
    )
  )
)
```

The point: The colored parentheses and indentation denotes the beginnings and endings of some *very large* lambda expressions, which simply denote everything that should be done with the thing that is being piped into it.

But wait, let's revisit the pipeline we just created:

```
["1", "2", "3"]        |> (fn L1 =>
map Int.fromString L1  |> (fn L2 =>
map Option.valOf L2    |> (fn L =>
foldr op+ 0 L          |> (fn total =>
List.length L          |> (fn len =>
total div len          )))))
```

It seems that every single function call is destined to be immediately piped into a lambda of what it must do next. What if we cut out the middleman, and define each function so it takes in that lambda directly?

We call such functions, that take in lambdas that their results are going to be piped into, **cool functions**.

```
fun mapCool f L k        = map f L |> k
fun foldrCool f acc L k = foldr f acc L |> k
fun lengthCool L k       = length L |> k
```

```
["1", "2", "3"]              |> (fn L1 =>
mapCool Int.fromString L1       (fn L2 =>
mapCool Option.valOf L2         (fn L =>
foldrCool op+ 0 L               (fn total =>
lengthCool L                    (fn len =>
total div len                   )))))
```

Please take a second to convince yourself that, via referential transparency, this is exactly equivalent to the previous code.

But all of this seems a little complicated. Why don't we just use a `let`?

```
let
  val L1    = ["1", "2", "3"]
  val L2    = map Int.fromString L1
  val L     = map Option.valOf L1
  val total = foldr op+ 0 L
  val len   = List.length L
in
  total div len
end
```

It's similarly readable.[2]

[2] Actually, massively more so.

What if we wanted to do this for a recursive function, though?

```
fun fact 0 = 1
  | fact n =
    let
      val rec_ans = fact (n - 1)
      val res = n * rec_ans
    in
      res
    end
```

Uh oh! We run into a problem that we saw several lectures ago...

This function is not tail recursive!

We make the recursive call to `fact`, and then multiply it by `n`. Actually, this is exactly equivalent to the ordinary `fact` function we would ordinarily write, just spread out on multiple lines.

We know the solution, of course. Let's write:

```
fun tfact 0 acc = acc
  | tfact n acc = tfact (n - 1) (acc * n)
```

Is it always straightforward, though?

Let's try `map`. For this one, we similarly cannot naively use a `let` for our intermediate computations.

```
fun tmap f [] acc = acc
  | tmap f (x::xs) acc = tmap f xs (f x :: acc)

fun map f L = tmap f L []
```

Seems good, right?

Warning Wrong. This is an incorrect implementation of `map`!

We see that `tmap Int.toString [1, 2, 3] []` ↪ `["3", "2", "1"]`! It reverses the list!

What can we do? Well, maybe all is not lost. It happens to be that we know how to implement a tail-recursive version of `rev`:

```
fun tmap_backwards f [] acc = acc
  | tmap_backwards f (x::xs) acc = tmap f xs (f x :: acc)

fun trev [] acc = acc
  | trev (x::xs) acc = trev xs (x::acc)

fun map f L = trev (tmap_backwards f L []) []
```

But this just becomes harder to reason about, and ugly as well. Our goal is that every function should admit a tail-recursive version. Can we make this process more natural?

How did we get here?

1. We wanted to write nicely sequenced operations that chained together
2. We wanted to be more explicit about our intermediate computations, because we might want to use them in later steps

One way to solve is by what we did earlier, having each function in our pipeline take in a lambda which contained the "next step" to be done.

These criteria are also satisfied with `let` expressions, except for the fact that in most cases, simply using a `let` ends up being non tail recursive!

Key This method of taking in a lambda of the "next step" provides a more straightforward way to translate a function to a tail recursive style. We call this **continuation-passing style**.

2 - Continuation-Passing Style

What is a **continuation**?

Def A **continuation** is a function taken in as an argument, which denotes what to do *after* the current computation.

It is named as such because it tells the function taking it in how to *continue* once it finishes its computation.

For instance, the declaration

```
fun mapCool f L k = map f L |> k
```

has the function `k` as a continuation, because it passes its return value directly to the continuation function.

The type of a function taking in a continuation changes in a predictable way.

For instance, if we want a function of type `int -> string -> bool` to take in a continuation, then its type would change from

```
int -> string -> bool
```

to

```
int -> string -> (bool -> 'a) -> 'a
```

Note The return type is polymorphic, because it depends on what the particular continuation that is passed in does!

For instance, suppose we wanted to write the following function in CPS:

```
add : int -> int -> int
```
REQUIRES: `true`
ENSURES: `add x y` $\cong$ `x + y`

```
fun add x y = x + y
```

The following would suffice:

```
addCPS : int -> int -> (int -> 'a) -> int
```
REQUIRES: `true`
ENSURES: `addCPS x y k` $\cong$ `k (x + y)`

```
fun addCPS x y k = k (x + y)
```

Is it always as simple as this, though? Suppose we were trying to make `fact` in CPS:

```
fun factCPS f L k = k (fact f L)
```

Warning This function is not tail recursive, nor is it CPS!

There's something wrong with our implementation. We do indeed call `k`, but we make another call to `fact`, which is a hugely non tail-recursive function! Moreover, it trivializes the problem.

Our goal here was to transform a non-tail-recursive function into a tail-recursive one. Just adding a trivial continuation like this doesn't make it tail recursive, however.

To convert a function like `fact` into CPS, we cannot rely on the definition of `fact` itself. We need to rewrite `fact` entirely!

Let's relate this to that idea of **cool functions** that we defined earlier, which are functions that pipe some result into the continuation they take in.

As we said, `factCPS` as we've defined here is most definitely not actually in CPS:

```
fun factCPS f L k = k (fact f L)
```

but it is a cool function, because it gives its result to `k`, its continuation.

Key Fact **CPS is cool**. That means all CPS functions are cool functions. But not all cool functions are CPS.

For defining recursive functions like `factCPS`, it turns out definitions like the above are on the right track, they just aren't cool *enough*. We will define a cooler function that maintains the property of tail recursion.

Here are the rules defining a function in CPS:

Def We say a function is in **continuation-passing** style if it fulfills the following criteria:

1. It takes in and uses continuations
2. It makes calls to other functions with continuations (including itself) as tail calls
3. It only calls continuations as tail calls

Key Fact **CPS is cool + tail recursive!**

This is what we had before, when we tried to write sequencing using `let`:

```
fun fact 0 = 1
  | fact n =
    let
      val rec_ans = fact (n - 1)
      val res = n * rec_ans
    in
      res
    end
```

How can we rewrite this in CPS?

For recursive functions, the process of converting to CPS is more involved.

The distinction here has to do with the fact that we have to make sure any recursive calls to the function happen last! Otherwise, our function will not be tail recursive, and therefore not in CPS.

The way to think about this is to draw a distinction between **writing down instructions** versus **remembering instructions**. In another sense, the difference is the distinction between **now** and **later**.

My friend Jonny and I are in a band, and every so often we want to play music together. I ask Jonny to go print out the sheet music for our latest song, and he returns to me with the music.

Unfortunately, the stack of papers is all out of order! I sort them so that the music is in the right order, and we play.

This happens a couple of times before I realize that the fact that I need to keep organizing the music after Jonny prints it for me is annoying. I need to drop what I'm doing and start sorting it, and the key problem is I need to *remember* to sort it! This takes up space in my brain.

A better way of doing things would be for me to, instead of **later** having to remember to sort the papers, to ask Jonny right **now** to sort the papers for me, and then bring them back to me in the right order.

```
let
  val rec_ans = fact (n - 1)
  val res = n * rec_ans
in
  res
end
```

Using `let` to make a few bindings illustrates the **later** mindset, which entails **remembering instructions**.

In this code, I need to first make a call to `fact (n - 1)`, and then *remember to multiply it afterwards*. This takes up space in the computer, because we need to do something after the `fact`! This is super not nice.

This means that after the recursive call, we **later** need to remember to do further work. This means we have to **remember instructions**.

But, if we just give the perspective a switch:

```
factCPS : int -> (int -> 'a) -> 'a
```
REQUIRES: `n >= 0`
ENSURES: `factCPS n k` evaluates to `k (fact n)`

```
fun factCPS 0 k = k 1
  | factCPS n k =
    factCPS (n - 1) (fn rec_ans =>
      let
        val rec_ans = rec_ans
        val res = n * rec_ans
      in
        k res
      end
    )
```

In this example, instead of executing the call to `factCPS` and then having work to do after, we make a tail call to `factCPS`!

The difference is that we put the work that must be done afterwards into a **continuation**, which is to say a lambda expression. This means that we don't need to do anything after the recursive call, but we *tell the recursive call what it needs to do after*, **now**.

Another way to think about it is that, by modifying the function that we pass into our CPS function, like `factCPS`, we are essentially treating the continuation as a **functional accumulator**, albeit one which accumulates instructions rather than data.

In this analogy, Jonny is our recursive call, and the sheet music is the value that we want it to return.

The choices are either to be **direct**, or to use CPS.

In a **direct-style** function, we have our recursive call return to us a value, which we then need to remember to do something to. We need to **remember instructions** to **later** execute on the value which it returns to us. In the case of `fact`, that is to multiply the recursive value by `n`.

In a **CPS** function, we **write down instructions** by encoding them into a lambda, and then we *give those instructions to the recursive call*. This way, we need to do no work on our part – the recursive call takes care of it for us. This means that instead of needing to remember instructions, we simply write it down **now**.

I've used the analogy of lambda expressions as instructions a few times now.

The idea is that a lambda expression is a list of steps to be done with a currently unknown input. The key observation is that everything in the body of a lambda expression **is not evaluated**. This means that writing down something like:

```
(fn onions => onions |> chop |> grill |> put sandwich)
```

is equivalent in intention to a list of instructions, which says:

- Take the onions
- Chop the onions
- Grill the onions
- Put the onions on a sandwich

After some extensionally equivalent refactoring, we end up with the CPS translation of `fact` as:

```
fun factCPS 0 k = k 1
  | factCPS n k =
      factCPS (n - 1) (fn rec_ans => k (n * rec_ans))
```

The final interpretation of this function is as one which, instead of computing the result of `fact` on `n - 1`, and then returning that value to be multiplied, we write down the *instruction* to multiply by `n` **now**, and carry that forward into the recursive call.

We also see that this function can easily be expressed in a way that resembles the pipelines we discussed at the beginning of this lecture!

Let's rewrite it slightly so that instead of the body being `k (n * rec_ans)`, it's the extensionally equivalent `n * rec_ans |> k`.

```
fun factCPS 0 k = k 1
  | factCPS n k =
      factCPS (n - 1)     (fn rec_ans =>
      n * rec_ans      |> k)
```

Let's do a trace to see how this function actually works!

link to asciinema video

One way to envision how this function actually works is that it *accumulates its continuation*.

Let's see how the continuation changes from line to line:

```
factCPS 3 k
```

```
factCPS 2     (fn res1 =>
3 * res1   |> k )
```

```
factCPS 1     (fn res2 =>
2 * res2   |> (fn res1 =>
3 * res1   |> k))
```

```
factCPS 0     (fn res3 =>
1 * res3   |> (fn res2 =>
2 * res2   |> (fn res1 =>
3 * res1   |> k)))
```

And now, what happens when we break down the continuation? We apply the giant lambda expression to the argument `1`, which is the same as piping `1` into it:

```
1          |> (fn res3 =>
1 * res3   |> (fn res2 =>
2 * res2   |> (fn res1 =>
3 * res1   |> k)))
```

```
1 * 1      |> (fn res2 =>
2 * res2   |> (fn res1 =>
3 * res1   |> k))
```

```
2 * 1       |> (fn res1 =>
3 * res1    |> k)
```

```
3 * 2       |> k
```

```
6           |> k
```

This notation might seem a little arcane, but it can help you understand how CPS is just a matter of consing onto something which looks like a list of instructions, and then consuming them from left-to-right.

Equipped with this understanding, we can see that this function:

```
fun mystery [] k       = k []
  | mystery (x::xs) k = mystery xs (fn res => x :: res |> k)
```

is just the identity function, because we only ever append to the end of the list of instructions we generate, and we break it down from left-to-right. This means we will cons on the oldest elements first, thus preserving our ordering.

Check your understanding Verify this by writing down the "list of instructions", in the form we just did, and seeing how it's destructed to preserve the original order!

3 - CPS Translation

Here's the general formula for how we can carry out CPS conversion on a function.

We are, given a function `f : t1 -> t2`, seeking its CPS version, which is `f_cps : t1 -> (t2 -> 'a) -> 'a` such that `f_cps x k` $\cong$ `k (f x)`.

1. For a function with return type `t`, add an extra continuation argument of type `t -> 'a`, and then change the return type to `'a`.
2. Call the continuation on every single return value of the function.
3. Suppose there is a recursive call to the function, which is the expression `e`. Change that to a new variable, let's say `rec_ans`.
4. Change the body of the recursive case to one which first performs `e (fn rec_ans => <body>)`, where the `<body>` is just the current body of the function.

Let's execute these translation steps on `map`. First, we start off with the vanilla implementation:

```
fun map f []       = []
  | map f (x::xs) =
      f x :: map f xs
```

Then, let's add in the continuation `k`:

```
fun map f [] k       = []
  | map f (x::xs) k =
      f x :: map f xs
```

Now, let's call the continuation `k` on every expression which is returned by the function:

```
fun map f [] k       = k []
  | map f (x::xs) k =
      k (f x :: map f xs)
```

Next, let's identify the recursive calls to `map`. In this case, our recursive call `e` is just `map f xs`:

```
fun map f [] k       = k []
  | map f (x::xs) k =
      k (f x :: map f xs)
```

Let's assume that we have the answer to the recursive call already, call it `rec_ans`, and replace `e` with it:

```
fun map f [] k       = k []
  | map f (x::xs) k =
      k (f x :: rec_ans)
```

Now, we take the recursive case and wrap it in a tail-recursive call to `e`, the expression we just replaced, except given a continuation binding `rec_ans`:

```
fun map f [] k       = k []
  | map f (x::xs) k =
      map f xs (fn rec_ans => k (f x :: rec_ans))
```

Now we have a complete, CPS version of `map`!

```
fun mapCPS f [] k       = k []
  | mapCPS f (x::xs) k =
      mapCPS f xs (fn rec_ans => k (f x :: rec_ans))
```

It is important to remember *why* this works in the first place.

All we are doing is making the binding of the recursive call to `map` explicit. Instead of leaving it somewhere nested in a big expression, to be used later, we *first* make the recursive call to the CPS-ified `map`, which is then *given instructions* on what to do with the recursive call's value.

The application of `k` to each returning expression is necessary to complete the "inductive handshake", to fulfill the promise that we pass whatever value we compute to the continuation.

4 - Control Flow

We can CPS-ify more interesting examples, that demonstrate the ability to express more nuanced control flow using continuations.

Consider the case of a function which is allowed to fail, i.e. returns a type `t option` for some type `t`, such as finding an element that satisfies a predicate in a tree:

```
fun search p Empty = NONE
  | search p (Node (L, x, R)) =
      if p x then
        SOME x
      else
        case search p L of
          NONE => search p R
        | SOME res => SOME res
```

When the recursive call is made to the function, we see that we have to handle two cases:

```
case search p L of
  NONE => search p R
| SOME res => SOME res
```

In essence, the function needs to "continue" from the recursive call in one of two cases:

- no information, in the `NONE` case
- a value of type `t`, for a `t tree`, in the `SOME` case

We can think of this as a *success* case and a *failure* case.

To make our reasoning more explicit, we will separate the logic of these cases into *two* continuations.

Instead of a single continuation of type `t option -> 'a`, we will instead have a **success continuation** of type `t -> 'a`, and a **failure continuation** of type `unit -> 'a`.

Instead of calling our continuation on an optional value to decide which case to branch on, we will simply call our failure continuation when we would otherwise return `NONE`, and call our success continuation on the value we would otherwise inject into `SOME`.

Let's clarify what our specifications should be, before and after our translation:

`search : ('a -> bool) -> 'a tree -> 'a option`
REQUIRES: `p` is total
ENSURES: `search p T` evaluates to the first element in `T` that satisfies `p`, in inorder traversal, else `NONE`

`searchCPS : ('a -> bool) -> 'a tree -> ('a -> 'b) -> (unit -> 'b) -> 'b`
REQUIRES: `p` is total
ENSURES:

$$\texttt{searchCPS p T sc fc} \cong \left\{ \begin{array}{ll} \texttt{sc x}, & \text{if } \texttt{search p T} \cong \texttt{SOME x} \\ \texttt{fc ()}, & \text{otherwise} \end{array} \right\}$$

Here's the general formula for how we can carry out CPS conversion on a function which returns an optional value:

1. For a function with return type `t option`, add two continuation arguments – one of type `t -> 'a` (the **success continuation**), and then one of type `unit -> 'a` (the **failure continuation**). Change the return type to `'a`.
2. For every return of `NONE`, call the failure continuation instead. For every return of `SOME x`, call the success continuation on `x` instead.
3. Suppose there is a `case` on a recursive call to the function. Replace the `case` with a call to the CPS-ified function itself, but with the success continuation changed to the code in the `SOME` case, and the failure continuation changed to the code in the `NONE` case.

Let's add in the success and failure continuations, `sc` and `fc`.

```
fun search p Empty sc fc = NONE
  | search p (Node (L, x, R)) sc fc =
      if p x then
        SOME x
      else
        case search p L of
          NONE => search p R
        | SOME res => SOME res
```

Now let's change the implicit returning values to explicit calls to `sc` and `fc`:

```
fun search p Empty sc fc = fc ()
  | search p (Node (L, x, R)) sc fc =
      if p x then
        sc x
      else
        case search p L of
          NONE => search p R
        | SOME res => sc res
```

We see one case where we dispatch on the return value of `search p L`.

Let's change that to instead be a call to `search`, except with the success and failure continuations corresponding to the code of each case.

First, let's identify each branch we are interested in:

```
fun search p Empty sc fc = fc ()
  | search p (Node (L, x, R)) sc fc =
      if p x then
        sc x
      else
        case search p L of
          NONE => search p R
        | SOME res => sc res
```

Now let's change it to an explicit recursive call:

```
fun search p Empty sc fc = fc ()
  | search p (Node (L, x, R)) sc fc =
      if p x then
        sc x
      else
        search p L
          (fn res => sc res)
          (fn () => search p R sc fc)
```

Now we have a complete CPS version of `search`!

```
fun searchCPS p Empty sc fc = fc ()
  | searchCPS p (Node (L, x, R)) sc fc =
      if p x then
        sc x
      else
        searchCPS p L
          (fn res => sc res)
          (fn () => searchCPS p R sc fc)
```

Continuation-passing style is merely a **different way of structuring computation**.

Instead of following implicit evaluation rules (left to right evaluation, outer then inner), we explicitly name and sequence every single computation, which makes our control flow more clear.

The process of CPS conversion that we've seen has been rather mechanical. This is intentional, because it is so mechanical that even a compiler can do it!

Regardless, understanding of CPS is an important skill. Being able to convert functions into CPS demonstrates mastery over the distinction between data as ordinary values versus data as function, and CPS-like code shows up in common applications, such as in the form of callbacks for web programming.

Thank you!