# Fun with higher-order functions: continuations 1

Jacob Thomas Errington

18 March 2019

# What's wrong with this function?

```
let rec append l1 l2 = match l1 with
| [] -> l2
| x :: xs -> x :: append xs l2
```

# What's wrong with this function?

```
let rec append l1 l2 = match l1 with
| [] -> l2
| x :: xs -> x :: append xs l2
```

It isn't *tail recursive*!

We could rewrite

```
let rec append l1 l2 = match l1 with
| [] -> l2
| x :: xs -> x :: append xs l2
```

We could rewrite

```
let rec append l1 l2 = match l1 with
| [] -> l2
| x :: xs -> x :: append xs l2
```

into

```
let rec append l1 l2 = match l1 with
| [] -> l2
| x :: xs ->
  let ys = append xs l2 in
  x :: ys
```

It's clear that the `::` (cons) is happening *after* the recursive call.

# So what?

- Tail recursive programs are amenable to *tail call optimization* (TCO).

# So what?

- Tail recursive programs are amenable to *tail call optimization* (TCO).
- TCO is a technique used in compilers to recycle *stack frames* when a call is the final expression to evaluate in a function. Such a call is called a tail call.

# So what?

- Tail recursive programs are amenable to *tail call optimization* (TCO).
- TCO is a technique used in compilers to recycle *stack frames* when a call is the final expression to evaluate in a function. Such a call is called a tail call.
- Stack memory is not very large (about 8 MiB), so non-tail recursive functions can only recurse so many times before they exhaust it.

# Uh-oh...

Are there functions we can't write tail recursively?

# Uh-oh...

Are there functions we can't write tail recursively?

If so, is functional programming (with recursion) a waste of time?

# Uh-oh...

Are there functions we can't write tail recursively?

If so, is functional programming (with recursion) a waste of time?

If so, why even live?

No.

All functions can be written tail recursively.

How?

Continuations!

# What is a continuation, formally?

### Definition

A continuation is a representation of the *execution state* of a program (e.g. the call stack) at a certain point in time.

# What is a continuation, formally?

Some languages provide *first-class* continuations, meaning that they provide constructs specifically for saving and restoring execution states.

# What is a continuation, formally?

Some languages provide *first-class* continuations, meaning that they provide constructs specifically for saving and restoring execution states.

e.g. `async/await` in C#, coroutines in Lua, callbacks/promises in JavaScript.

# What is a continuation, formally?

In OCaml, we will simply use functions to implement continuations.

# In practice...

- A continuation is an extra argument passed to a function.
- This argument is a *function*; it acts as a *generalized accumulator*.
- It encodes what the function should do next, when it has computed its result.

Demo!

# Step-by-step, how does this work?

Consider this example; it's the base case.

```
append_k [] [1;2] (fun x -> x)
```

# Step-by-step, how does this work?

Consider this example; it's the base case.

```
append_k [] [1;2] (fun x -> x)
(fun x -> x) [1;2]
```

# Step-by-step, how does this work?

Consider this example; it's the base case.

```
append_k [] [1;2] (fun x -> x)
(fun x -> x) [1;2]
[1;2]
```

Nothing too strange.

# Step-by-step, how does this work?

Now let's try with a nonempty list. This is trickier!

# Step-by-step, how does this work?

Now let's try with a nonempty list. This is trickier!

Recall the definition.

```
let rec append_k l1 l2 k = match l1 with
  | [] -> k l2
  | x :: xs ->
     append_k xs l2 (fun r -> k (x :: r))
```

# Step-by-step, how does this work?

Now let's try with a nonempty list. This is trickier!

Recall the definition.

```
let rec append_k l1 l2 k = match l1 with
  | [] -> k l2
  | x :: xs ->
     append_k xs l2 (fun r -> k (x :: r))
```

```
append_k [1;2] ls id
```

# Step-by-step, how does this work?

Now let's try with a nonempty list. This is trickier!

Recall the definition.

```
let rec append_k l1 l2 k = match l1 with
  | [] -> k l2
  | x :: xs ->
     append_k xs l2 (fun r -> k (x :: r))
```

```
append_k [1;2] ls id
append_k [2] ls (fun r -> id (1 :: r))
```

# Step-by-step, how does this work?

Now let's try with a nonempty list. This is trickier!

Recall the definition.

```
let rec append_k l1 l2 k = match l1 with
  | [] -> k l2
  | x :: xs ->
     append_k xs l2 (fun r -> k (x :: r))
```

```
append_k [1;2] ls id
append_k [2] ls (fun r -> id (1 :: r))
append_k [] ls (fun r' -> (fun r -> id (1 :: r)) (2 :: r'))
```

A "stack" of pending operations has been built up explicitly in the continuation!

# What's next?

Discuss with the person beside you what the next step should be after this!

```
append_k [] ls (fun r' -> (fun r -> id (1 :: r)) (2 :: r'))
```

# What's next?

Discuss with the person beside you what the next step should be after this!

```
append_k [] ls (fun r' -> (fun r -> id (1 :: r)) (2 :: r'))
```

1. `append_k [] ls (fun r' -> id (1 :: (2 :: r'))) ls`
2. `(fun r' -> (fun r -> id (1 :: r)) (2 :: r')) ls`
3. `id (1 :: 2 :: ls)`
4. `(fun r' -> id (1 :: (2 :: r')) ls`

Vote at `http://pingo.coactum.de/087757` with your phone or your computer!

# Just a few more steps...

```
(fun r' -> (fun r -> id (1 :: r)) (2 :: r')) ls
```

# Just a few more steps...

```
(fun r' -> (fun r -> id (1 :: r)) (2 :: r')) ls
(fun r -> id (1 :: r)) (2 :: ls)
```

# Just a few more steps...

```
(fun r' -> (fun r -> id (1 :: r)) (2 :: r')) ls
(fun r -> id (1 :: r)) (2 :: ls)
id (1 :: (2 :: ls))
```

# Just a few more steps...

```
(fun r' -> (fun r -> id (1 :: r)) (2 :: r')) ls
(fun r -> id (1 :: r)) (2 :: ls)
id (1 :: (2 :: ls))
1 :: 2 :: ls
```

# Remark: performance

`append` will outperform `append_k` on small lists. On (very) large lists, `append` will crash whereas `append_k` will run.

# Remark: performance

`append` will outperform `append_k` on small lists. On (very) large lists, `append` will crash whereas `append_k` will run.

Both `append` and `append_k` use $O(n)$ memory, but it's the *type* of memory used that is different.

| | |
|---|---|
| append | uses the *stack* which is not very big (8 MiB) |
| append_k | uses the *heap* which is plentiful (several GiB) |

# Remark: performance

`append` will outperform `append_k` on small lists. On (very) large lists, `append` will crash whereas `append_k` will run.

Both `append` and `append_k` use $O(n)$ memory, but it's the *type* of memory used that is different.

| | |
|---|---|
| append | uses the *stack* which is not very big (8 MiB) |
| append_k | uses the *heap* which is plentiful (several GiB) |

However, the use of continuations in the form of *closures* (functions capturing an environment) incurs an extra time and space penality.

# Recap: how to convert to continuation-passing style

1. Change the type signature: add the continuation.
   e.g. `append : 'a list -> 'a list -> 'a list`.
   `append_k : 'a list -> 'a list -> ('a list -> 'r) -> 'r`.

# Recap: how to convert to continuation-passing style

1. Change the type signature: add the continuation.
   e.g. `append : 'a list -> 'a list -> 'a list`.
   `append_k : 'a list -> 'a list -> ('a list -> 'r) -> 'r`.
2. All the work should happen after the recursive call gets moved into the continuation.
   e.g.

   ```
   | x :: xs ->
     let ys = append xs l2 in
     x :: ys
   ```

   becomes

   ```
   | x :: xs ->
     append_k xs l2
       (fun ys -> x :: ys)
   ```

# Advanced control flow with continuations

# Recap: dealing with failure

You can use the `option` type to model computations which may fail, e.g. finding a value in a tree satisfying a predicate.

```
find : ('a -> bool) -> 'a tree -> 'a option
```

Since there may be no such value satisfying the function `'a -> bool`, we return an `'a option` so that we can return `None` to signify that the lookup failed.

# Recap: dealing with failure

You can use the `option` type to model computations which may fail, e.g. finding a value in a tree satisfying a predicate.

```
find : ('a -> bool) -> 'a tree -> 'a option
```

Since there may be no such value satisfying the function `'a -> bool`, we return an `'a option` so that we can return `None` to signify that the lookup failed.

We can use the general recipe to convert this function to continuation-passing style, but it turns out that we can do even better!

Demo!