Programming Languages and Paradigms COMP 302

Instructor: Jacob Thomas Errington
School of Computer Science
McGill University

Application of higher-order functions: continuations

Let's talk performance

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

Discuss with the person beside you the performance characteristics of this function.

- What is its time complexity?
- Its space complexity?

Let's talk performance

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

Let's talk performance

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

Running time: linear in `l1` – $O(n)$

Let's talk performance

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

Running time: linear in `l1` – $O(n)$

Memory usage: linear in `l1` – $O(n)$

Why linear memory usage?

We could rewrite

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

Why linear memory usage?

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
```

Why linear memory usage?

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 returns.

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

How would we even rewrite this using an accumulator???

All functions can be written tail recursively.

Idea: use a *higher-order* accumulator

- This accumulator, instead of storing list items, stores pending operations.
- That's normally what the call stack does for us!

Rewriting append tail-recursively

demo

Step-by-step, how does this work?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

Consider this example; it's the base case.

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

Step-by-step, how does this work?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

Consider this example; it's the base case.

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

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

Step-by-step, how does this work?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

Consider this example; it's the base case.

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

Nothing too strange.

Step-by-step, how does this work?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

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

Step-by-step, how does this work?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

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

`append_tr [1;2] ls id`

Step-by-step, how does this work?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

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

`append_tr [1;2] ls id`

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

Step-by-step, how does this work?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

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

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

Step-by-step, how does this work?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

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

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

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

What's next?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

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

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

What's next?

```
let rec append_tr l1 l2 return =
  match l1 with
  | [] -> return l2
  | x :: xs ->
    append_tr xs l2
      (fun ys -> return (x :: ys))
```

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

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

1. `append_tr [] 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`

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
```

More generally

The “return function” we introduced is an example of a continuation.

More generally

The “return function” we introduced is an example of a continuation.

Continuations

In general, a continuation is a representation of the *execution state* of a program at a certain point in time.

More generally

The "return function" we introduced is an example of a continuation.

Continuations

In general, a continuation is a representation of the *execution state* of a program at a certain point in time.

Usually, the execution state we want to represent and manipulate ourselves is the *call stack*.

More generally

The "return function" we introduced is an example of a continuation.

Continuations

In general, a continuation is a representation of the *execution state* of a program at a certain point in time.

Usually, the execution state we want to represent and manipulate ourselves is the *call stack*.
Specifically: what work is left to do? In other words; "what happens when we return?" Continuations let us manipulate *what happens next* directly.

In other languages

Some languages provide *second-class* continuations at least in some form: they provide constructs specifically for saving, manipulating, and restoring execution states.

- Async programming using `async`/`await` (C#, JavaScript)
- Coroutines and generators (JavaScript, Lua, Python)

In other languages

Some languages provide *second-class* continuations at least in some form: they provide constructs specifically for saving, manipulating, and restoring execution states.

- Async programming using `async/await` (C#, JavaScript)
- Coroutines and generators (JavaScript, Lua, Python)

Some languages provide *first-class* continuations: you can save execution states, and manipulate them just like you would any other function. Calling the function re-instates the saved execution state.

- Scheme and some other Lisps
- sml/NJ (similar to OCaml)

Remark: performance

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

Remark: performance

`append` will outperform `append_tr` on small lists. On (very) large lists, `append` will crash whereas `append_tr` will run.
Both `append` and `append_tr` 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_tr	uses the *heap* which is plentiful (several GiB)

Remark: performance

`append` will outperform `append_tr` on small lists. On (very) large lists, `append` will crash whereas `append_tr` will run. Both `append` and `append_tr` 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_tr
: uses the *heap* which is plentiful (several GiB)

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

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`.
   becomes
   `append_tr : '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`.
   becomes
   `append_tr : 'a list -> 'a list -> ('a list -> 'r) -> 'r`.
2. Move all the work that should happen after the recursive call into the continuation.
   e.g.

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

   becomes

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

   and returning becomes an explicit call to the continuation.

Conclusion

Higher-order functions allow us to:

- express concise generic algorithms;
- directly manipulate control flow.

Next time: creatively using this control flow manipulation.