

Foundations of Computer Science

Lazy Lists: pipelines and streams

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

Question 1: What is the type of this function?

```
In[1]: let cf y x = y
```

```
Out[1]: val cf : 'a -> 'b -> 'a = <fun>
```

Question 2: What does `cf y` return?

Question 3: We have the following: `let add a b = a + b`

Use a partial application of `add` to define an increment function:

Question 1: What is the type of this function?

```
In[1]: let cf y x = y
```

```
Out[1]: val cf : 'a -> 'b -> 'a = <fun>
```

Question 2: What does `cf y` return?

Question 3: We have the following: `let add a b = a + b`

Use a partial application of `add` to define an increment function:

Question 1: What is the type of this function?

```
In[1]: let cf y x = y
```

```
Out[1]: val cf : 'a -> 'b -> 'a = <fun>
```

Question 2: What does `cf y` return?

Question 3: We have the following: `let add a b = a + b`

Use a partial application of `add` to define an increment function:

Question 1: What is the type of this function?

```
In[1]: let cf y x = y
```

```
Out[1]: val cf : 'a -> 'b -> 'a = <fun>
```

Question 2: What does `cf y` return?

It returns a constant function.

Question 3: We have the following: `let add a b = a + b`

Use a partial application of `add` to define an increment function:

Question 1: What is the type of this function?

```
In[1]: let cf y x = y
```

```
Out[1]: val cf : 'a -> 'b -> 'a = <fun>
```

Question 2: What does `cf y` return?

It returns a constant function.

Question 3: We have the following: `let add a b = a + b`

Use a partial application of `add` to define an increment function:

```
let increment = add 1
```

Question 4: What is the type of `f`?

```
let f x y z = x z (y z)
```

```
val f : ?
```

Question 4: What is the type of f?

let f x y z = x z (y z)




f has **function type**

val f : ?x -> ?y -> ?z -> ?r

Question 4: What is the type of f?

let f x y z = x z (y z)



y has type ?z -> ?s

val f : ?x -> (?z -> ?s) -> ?z -> ?r

Question 4: What is the type of f?

let f x y z = x z (y z)

x has type ?z -> ?s -> ?r

val f : (?z -> ?s -> ?r) -> (?z -> ?s) -> ?z -> ?r

Question 4: What is the type of `f`?

```
let f x y z = x z (y z)
```

No more information: **generalize**

```
val f : ('z -> 's -> 'r) -> ('z -> 's) -> 'z -> 'r
```

Question 5: Is this function tail-recursive? Why?

```
let rec exists p = function  
| [] -> false  
| x::xs -> (p x) || exists p xs
```

Question 5: Is this function tail-recursive? Why?

```
let rec exists p = function  
| [] -> false  
| x::xs -> (p x) || exists p xs
```

It **is** tail-recursive:

```
let rec exists p = function  
| [] -> false  
| x::xs -> (p x) || ((exists[@ocaml.tailcall]) p xs)
```

Question 5: Is this function tail-recursive? Why?

```
let rec exists p = function
| [] -> false
| x::xs -> (p x) || exists p xs
```

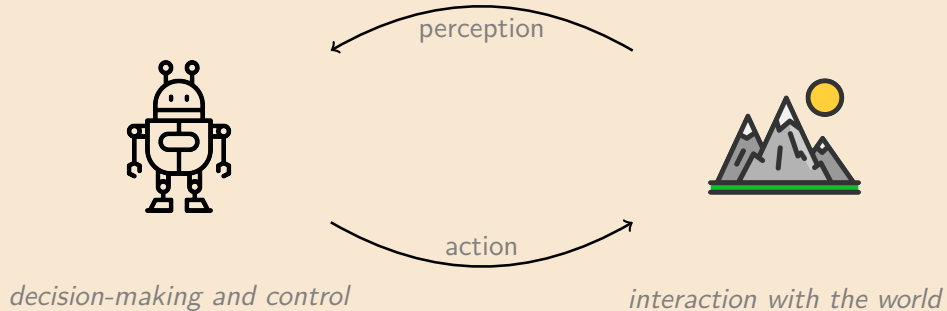
It **is** tail-recursive:

```
let rec exists p = function
| [] -> false
| x::xs -> (p x) || ((exists[@ocaml.tailcall]) p xs)
```

Why it is tail-recursive:

```
let rec exists p = function
| [] -> false
| x::xs -> if p x then true else exists p xs
```

Example: perception-action loops (basic building block of autonomy)



```
while(true)
  get sensor data
  act upon sensor data
  repeat
```

Exhaustive search



searching for keywords

Data processing



image processing

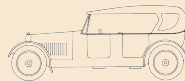
Sequential programs

Control tasks



robot navigation

Resource allocation



mobility on demand

Reactive programs



Produce sequence of items

Filter sequence in stages

Consume results as needed

Lazy lists join the stages together

Lists of possibly **infinite length**

elements **computed upon demand**

avoids waste if there are many solutions

infinite objects are a useful abstraction

In OCaml: implement laziness by delaying evaluation of the tail

In OCaml: "streams" means input/output channels, so we use term '**sequences**'

The type unit has one element: empty tuple ()

What use is ()?

- () can appear in data-structures (e.g., unit-valued dictionary)
- () can be the argument of a function
- () can be the argument or result of a procedure (later in course)

Behaves as a **tuple**, is a **constructor**, and allowed in **pattern matching**:

```
let f () = ...
```

```
let f = function  
| () ->
```

Delayed evaluation:

```
fun () -> E
```

```
In[2]: type 'a seq =
        | Nil
        | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
```

```
Line 1, characters 9-26:
```

```
1 | let head (Cons (x, _)) = x;;
```

```
*****
```

```
Warning 8: this pattern-matching is not exhaustive.
```

```
Here is an example of a case that is not matched:
```

```
Nil
```

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply xf to () to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
```

```
Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```

```
In[2]: type 'a seq =
        | Nil
        | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
```

```
Line 1, characters 9-26:
```

```
1 | let head (Cons (x, _)) = x;;
```

```
*****
```

```
Warning 8: this pattern-matching is not exhaustive.
```

```
Here is an example of a case that is not matched:
```

```
Nil
```

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply `xf` to `()` to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
```

```
Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```

```
In[2]: type 'a seq =
        | Nil
        | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
Line 1, characters 9-26:
1 | let head (Cons (x, _)) = x;;
   ^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Nil
```

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply `xf` to `()` to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```

```
In[2]: type 'a seq =
        | Nil
        | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
```

```
Line 1, characters 9-26:
```

```
1 | let head (Cons (x, _)) = x;;
```

```
*****
```

```
Warning 8: this pattern-matching is not exhaustive.
```

```
Here is an example of a case that is not matched:
```

```
Nil
```

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply `xf` to `()` to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
```

```
Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```

```
In[2]: type 'a seq =
      | Nil
      | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
      Line 1, characters 9-26:
      1 | let head (Cons (x, _)) = x;;
          ^^^^^^^^^^^^^^^^^^^^^
```

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Nil

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply `xf` to `()` to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
      Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```



```
In[2]: type 'a seq =
      | Nil
      | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
Line 1, characters 9-26:
1 | let head (Cons (x, _)) = x;;
      ^^^^^^^^^^^^^^^^^^^
```

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Nil

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply xf to () to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```

```
In[2]: type 'a seq =
      | Nil
      | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
      Line 1, characters 9-26:
      1 | let head (Cons (x, _)) = x;;
          ^^^^^^^^^^^^^^^^^^^^^
```

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Nil

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply xf to () to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
      Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```

```
In[2]: type 'a seq =
      | Nil
      | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
      Line 1, characters 9-26:
      1 | let head (Cons (x, _)) = x;;
          ^^^^^^^^^^^^^^^^^^^^^
```

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Nil

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply xf to () to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
      Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```

```
In[2]: type 'a seq =
        | Nil
        | Cons of 'a * (unit -> 'a seq)
```

```
Out[2]: type 'a seq = Nil | Cons of 'a * (unit -> 'a seq)
```

```
In[3]: let head (Cons (x, _)) = x
Line 1, characters 9-26:
1 | let head (Cons (x, _)) = x;;
      ^^^^^^^^^^^^^^^^^^^
```

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Nil

```
Out[3]: val head : 'a seq -> 'a = <fun>
```

Apply xf to () to evaluate:

```
In[4]: let tail (Cons (_, xf)) = xf ()
Warning: (similar warning elided)
```

```
Out[4]: val tail : 'a seq -> 'a seq = <fun>
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: - : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: - : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: - : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: - : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```


The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: - : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: ~ : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: ~ : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: ~ : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: - : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: - : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```

The Infinite Sequence $k, k + 1, k + 2, \dots$

```
In[5]: let rec from k = Cons (k, fun () -> from (k + 1))
```

```
Out[5]: val from : int -> int seq = <fun>
```

```
In[6]: let it = from 1
```

```
Out[6]: val it : int seq = Cons (1, <fun>)
```

```
In[7]: let it = tail it
```

```
Out[7]: val it : int seq = Cons (2, <fun>)
```

```
In[8]: tail it
```

```
Out[8]: - : int seq = Cons (3, <fun>)
```

Recall (force the evaluation):

```
In[9]: let tail (Cons(_, xf)) = xf ()
```

```
Out[9]: val tail : 'a seq -> 'a seq
```