Foundations of Computer Science
Lecture #9: Sequences, or Lazy Lists

Anil Madhavapeddy
25th October 2023
Question 1: What is the type of this function?

```ml
let cf y x = y;;
```

```
Out: 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:

```ml
In : let increment = ???
```

```ml
In : let increment = add 1;;
```
**What is the type of \( f \)?**

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

**Step 1:** analyze the right-hand side expression

function

- type \( z \) : 'a
- return-type \( y \) : 'b
- return-type \( x \) : 'c

**Step 2:** what are the unknown types?

- input-type \( y \) : 'a
- input-type \( x \) : 'a -> 'b

**Step 3:** set those types.

- type \( y \) : 'a -> 'b
- type \( x \) : 'a -> 'b -> 'c
- type \( z \) : 'a

**Step 4:** infer the input types.

**Step 5:** infer all types.

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

**Step 6:** infer function type.

```
val f : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```
Question 4: Is this function tail-recursive? Why?

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

It is...

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

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

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

decision-making and control

perception

action

interaction with the world
Sequential programs - examples include:

- Exhaustive search
  - search a book for a keyword
  - search a graph for the optimal path
- Data processing
  - image processing (enhance / compress)
  - outlier removal / de-noise

Reactive programs - examples include:

- Control tasks
  - flying a plane
  - robot navigation (obstacle avoidance)
- Resource allocation
  - computer processor
  - Mobility-on-Demand (e.g. Uber)
Two types of program can be distinguished. A sequential program accepts a problem to solve, processes for a while, and finally terminates with its result. A typical example is the huge numerical simulations that are run on supercomputers. Most of our ML functions also fit this model.

At the other extreme are reactive programs, whose job is to interact with the environment. They communicate constantly during their operation and run for as long as is necessary. A typical example is the software that controls many modern aircraft. Reactive programs often consist of concurrent processes running at the same time and communicating with one another.

Concurrency is too difficult to consider in this course, but we can model pipelines such as that shown above. The Producer represents one or more sources of data, which it outputs as a stream. The Filter stages convert the input stream to an output stream, perhaps consuming several input items to yield a single output item. The Consumer takes as many elements as necessary. The Consumer drives the pipeline: nothing is computed except in response to its demand for an additional datum. Execution of the Filter stages is interleaved as required for the computation to go through. The program sets up data dependencies but has no clear idea of what happens when. We have the illusion of concurrent computation.

The Unix operating system provides similar ideas through its pipes that link processes together. In ML, we can model pipelines using lazy lists.
Lazy Lists — or Streams

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*’ reserved for input/output channels, so we use term ‘*sequences*’
The **type** `unit` has one element: empty tuple `()`

**Uses:**
- 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 (seen later in course)

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

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

Expression $E$ not evaluated until the function is applied:

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

*fun notation enables delayed evaluation!*
Lazy Lists in OCaml

type 'a seq =
| Nil
| Cons of 'a * (unit -> 'a seq)

let head (Cons (x, _)) = x
# val head : 'a seq -> 'a = <fun>
Lazy Lists in OCaml

```ocaml
type 'a seq =
 | Nil
 | Cons of 'a * (unit -> 'a seq)

let head (Cons (x, _)) = x
# val head : 'a seq -> 'a = <fun>

let tail (Cons (_, xf)) = xf ()
# val tail : 'a seq -> 'a seq = <fun>
```
Lazy Lists in OCaml

```ocaml
type 'a seq =
  | Nil
  | Cons of 'a * (unit -> 'a seq)

let head (Cons (x, _)) = x
# val head : 'a seq -> 'a = <fun>

let tail (Cons (_, xf)) = xf ()
# val tail : 'a seq -> 'a seq = <fun>
```

Cons \((x, xf)\) has head \(x\) and tail function \(xf\)

apply \(xf\) to () to evaluate
The Infinite Sequence, \( k, k+1, k+2, \ldots \)

let rec from k = Cons (k, fun () -> from (k + 1));;
val from : int -> int seq = <fun>

let it = from 1;;
val it : int seq = Cons (1, <fun>)

let it = tail it;;
val it : int seq = Cons (2, <fun>)

tail it;;
- : int seq = Cons (3, <fun>)

Recall:
let tail (Cons(_, xf)) = xf ();;
# val tail : 'a seq -> 'a seq
force the evaluation
let rec get n s =
  if n = 0 then []
  else
    match s with
    | Nil -> []
    | Cons (x, xf) -> x :: get (n - 1) (xf ())

Get the first $n$ elements as a list

$xf()$ forces evaluation
get 2 (from 6)
  \rightarrow get 2 (Cons (6, fun () -> from (6 + 1)))
  \rightarrow 6 :: get 1 (from (6 + 1))
  \rightarrow 6 :: get 1 (Cons (7, fun () -> from (7 + 1)))

  \rightarrow 6 :: 7 :: get 0 (from (7 + 1))

  \rightarrow 6 :: 7 :: get 0 (Cons (8, fun () -> from (8 + 1)))
  \rightarrow 6 :: 7 :: []
  \rightarrow [6; 7]
Joining Two Sequences

let rec appendq xq yq =
  match xq with
  | Nil -> yq
  | Cons (x, xf) ->
    Cons (x, fun () -> appendq (xf ()) yq)
Joining Two Sequences

```ocaml
let rec appendq xq yq = 
    match xq with 
    | Nil -> yq 
    | Cons (x, xf) -> 
      Cons (x, fun () -> appendq (xf ()) yq)

A fair alternative...

let rec interleave xq yq = 
    match xq with 
    | Nil -> yq 
    | Cons (x, xf) -> 
      Cons (x, fun () -> interleave yq (xf ()))
```
let rec filter p = function
| []  -> []
| x::xs ->
  if p x then
    x :: filter p xs
  else
    filter p xs
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>

We want:
val filterq : ('a -> bool) -> 'a seq -> 'a seq = <fun>
Functionals for Lazy Lists

filtering

let rec filterq p = function
  | Nil -> Nil
  | Cons (x, xf) ->
    if p x then
      Cons (x, fun () -> filterq p (xf ()))
    else
      filterq p (xf ())

let rec iterates f x =
  Cons (x, fun () -> iterates f (f x))

The infinite sequence \(x, f(x), f(f(x)), \ldots\)

let rec filterq p = function
  | Nil -> Nil
  | Cons (x, xf) ->
    if p x then
      Cons (x, fun () -> filterq p (xf ()))
    else
      filterq p (xf ())

let rec iterates f x =
  Cons (x, fun () -> iterates f (f x))

val filterq : ('a -> bool) -> 'a seq -> 'a seq = <fun>
val iterates : ('a -> 'a) -> 'a -> 'a seq = <fun>
Example:

val filterq : ('a -> bool) -> 'a seq -> 'a seq
val iterates : ('a -> 'a) -> 'a -> 'a seq

> let myseq = iterates (fun x -> x + 1) 1;;
# val myseq : int seq = Cons (1, <fun>)

> filterq (fun x -> x = 1) myseq;;
# - : int seq = Cons (1, <fun>)

> filterq (fun x -> x = 100) myseq;;
# - : int seq = Cons (100, <fun>)

> filterq (fun x -> x = 0) myseq;;
......
Reusing Functionals for Lazy Lists

Same Examples, but with no new functions:

> succ;;
- : int -> int = <fun>
> succ 1;;
- : 2 = int
> (=) 1 2
- : bool = false

> let myseq = iterates succ 1;;
val myseq : int seq = Cons (1, <fun>)
> filterq ((=) 1) myseq;;
- : int seq = Cons (1, <fun>)
> filterq ((=) 100) myseq;;
- : int seq = Cons (100, <fun>)
> filterq ((=) 0) myseq;;
......
Functionals for Lazy Lists

Example:

```
val filterq : ('a -> bool) -> 'a seq -> 'a seq
val iterates : ('a -> 'a) -> 'a -> 'a seq
val get : int -> 'a seq -> 'a list

> val myseq = iterates (fun x -> x + 1) 1;;
val myseq : int seq Cons (1, <fun>)

> let it = filterq (fun x -> x mod 2 = 0) myseq;;
val it : int seq = Cons (2, <fun>)

> get 5 it;;
- : int list = [2; 4; 6; 8; 10]
```
Numerical Computations on Infinite Sequences

\[ \text{find } \sqrt{a} \]

let next a x = (a /. x +. x) /. 2.0
Aside: Newton-Raphson Method

Series is:

\[ x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \]
\[ x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \]
\[ x_3 = \vdots \]
\[ x_4 = \vdots \]
\[ x_5 = \vdots \]

So if we want to find \( \sqrt{k} \) we use:

\[ x^2 = k \]
\[ f(x) = x^2 - k \]
\[ f'(x) = 2x \]
Aside: Newton-Raphson Method

Series is:

\[ x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \]
\[ x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \]
\[ x_3 = \vdots \]
\[ x_4 = \vdots \]
\[ x_5 = \vdots \]

So if we want to find \( \sqrt{k} \) we use:

\[ x^2 = k \]
\[ f(x) = x^2 - k \]
\[ f'(x) = 2x \]

\[ x_{n+1} = \frac{1}{2} \left( x_n + \frac{k}{x_n} \right) \]
Numerical Computations on Infinite Sequences

find sqrt(a) \[ x_n \]

let next a x = (a /. x +. x) /. 2.0

Close enough?

let rec within eps = function
| Cons (x, xf) ->
  match xf () with
| Cons (y, yf) ->
  if abs_float (x -. y) <= eps then y
  else within eps (Cons (y, yf))

\[ x_{n+1} = \frac{1}{2} \left( x_n + \frac{k}{x_n} \right) \]
Numerical Computations on Infinite Sequences

\[ \text{find } \sqrt{a} \]

let next a x = (a /. x +. x) /. 2.0

Close enough?

let rec within eps = function
| Cons (x, xf) ->
  match xf () with
  | Cons (y, yf) ->
    if abs_float (x -. y) <= eps then y
  else within eps (Cons (y, yf))

Square Roots!

let root a = within 1e-6 (iterates (next a) 1.0)

> root 3.0;;
- : float = 1.73205080756887719