Foundations of Computer Science
Lecture #8: Currying

Anil Madhavapeddy

2022-2023

Warm-Up

Question 1: How many arguments does this function have?

```
let rec append = function
| ([], ys)      -> ys
| (x::xs, ys) -> x :: append (xs,ys)
```

One (the argument is a tuple)

Question 2: What property does an inorder conversion of a binary tree to a list preserve?

List will be sorted

Question 3: What is the depth of a balanced binary search tree with *n* elements?

O (log n)

Functions as Values

In OCaml, functions can be

- passed as *arguments* to other functions,
- returned as *results*,
- put into lists, tree, etc.:

say "lambda"

```
[(fun n -> n * 2); (fun n -> n * 3); (fun k -> k + 1)];;
- : (int -> int) list = [<fun>; <fun>; <fun>]
```

- but **not** tested for equality.

Functions without Names

`fun x ->` E is the function f such that $f(x) = E$

The function `(fun n -> n * 2)` is a *doubling function*.

```
(fun n -> n * 2);;
- : int -> int = <fun>
```

```
(fun n -> n * 2) 17;;
- : int = 34
```

Functions without Names

```
In : (fun n -> n * 2) 2;;
Out: - : int = 4
```

… can be given a name by a `let` declaration

```
In :  let double = fun n -> n * 2;;
Out:  val double : int -> int = <fun>

In :  let double n = n * 2;;
Out:  val double : int -> int = <fun>
```

In both cases:

```
In : double 2;
Out: - : int = 4
```

Functions without Names

`function` can be used for pattern-matching:

`function` P_1 **->** E_1 | … | P_n **->** E_n

for example:

```
function 0 -> true | _ -> false
```

which is equivalent to:

```
fun x -> match x with 0 -> true | _ -> false

let is_zero = fun x -> match x with 0 -> true | _ -> false

let is_zero = function 0 -> true | _ -> false
```

Curried Functions

- Consider that a function can only have **one** argument
- Two options for **multiple** arguments:
  1. tuples (e.g., pairs) *[as seen in previous lectures]*
  2. a function that returns another function as a result

     → this is called ***currying*** (after H. B. Curry) [1]

- Currying: expressing a function taking multiple arguments as **nested functions**.

[1] Credited to Schönfinkel, but *Schönfinkeling* didn't catch on…

Curried Functions

Taking multiple arguments as **nested** functions, so, instead of:

```
In :   fun (n, k) -> n * 2 + k;;
Out:   - : int * int -> int = <fun>
```

We can **nest** the `fun`-notation:

```
In :   let it = fun k -> (fun n -> n * 2 + k);;
Out:   val it : int -> int -> int = <fun>
```

k n

```
In :   it 1 3;
Out: - : int = 7
```

Curried Functions

A *curried function* returns another function as its *result*.

```
let prefix = (fun a -> (fun b -> a ^ b))
val prefix : string -> string -> string = <fun>
```

`prefix` yields functions of type `string -> string`.

```
 let promote = prefix "Professor ";;
 val promote : string -> string = <fun>
```

```
promote "Mopp";;
- : string = "Professor Mopp"
```

Shorthand for Curried Functions

A function-returning function is just a *function of two arguments*

A function over pairs has type $(\sigma 1 \times \sigma 2) \to \tau$.
A curried function has type $\sigma 1 \to (\sigma 2 \to \tau)$.

This curried function is nicer than nested `fun` binders:

```
let prefix a b = a ^ b;;

val prefix : string -> (string -> string)
```

the symbol -> associates to the right

Syntax:

<u>`fun`</u> x_1 x_2 … x_n `->` E <u>`let`</u> `f` x_1 x_2 … x_n `=` E

```
let dub = prefix "Lady ";;
val dub : string -> string = <fun>
```

Curried functions allows *partial application* (to the first argument).

Partial Application: A Curried Insertion Sort

Key question: How to generalize <= to any data type?

```
let rec insort lessequal =
  let rec ins = function
  | x, [] -> [x]
  | x, y::ys ->
      if lessequal x y then x::y::ys
      else y :: ins (x, ys)
  in
  let rec sort = function
  | [] -> []
  | x::xs -> ins (x, sort xs)
  in
    sort
```

lessequal sort

```
val insort : ('a -> 'a -> bool) ->('a list -> 'a list)
```

IN OUT

Partial Application: A Curried Insertion Sort

Note: `(<=)` denotes comparison operator as a function

```
In : insort (<=) [5; 3; 9; 8];;
Out: - : int list = [3; 5; 8; 9]

In : insort (>=) [5; 3; 9; 8];;
Out: - : int list = [9; 8; 5; 3]

In : insort (<=) ["bitten"; "on"; "a"; "bee"];;
Out: - : string list = ["a"; "bee"; "bitten"; "on"]
```

map: the ‘Apply to All’ Functional

note: built-in as List.map

```
let rec map f = function
| []    -> []
| x::xs -> (f x) :: map f xs
```

```
In : map (fun s -> s ^ "ppy");
Out: -: string list -> string list = <fun>
```

```
In : map (fun s -> s ^ "ppy") ["Hi"; "Ho"];;
Out: - : string list = ["Hippy"; "Hoppy"]

In : map (map double) [[1]; [2; 3]];;
Out: - : int list list = [[2]; [4; 6]]
```

Example: Matrix Transpose

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}^T = \begin{pmatrix} a & d \\ b & e \\ c & f \end{pmatrix}$$

```
let rec transp = function
| [] :: _ -> []
| rows  -> (map List.hd rows) ::
           (transp (map List.tl rows))
```

Example: Matrix Transpose

```
let rec transp = function
| [] :: _ -> []
| rows  -> (map List.hd rows) ::
           (transp (map List.tl rows))


In : let rows = [[1; 2; 3]; [4; 5; 6]];;

In : List.hd;;
Out: - : 'a list -> ‘a = <fun>
In : transp;
Out: - : 'a list list -> 'a list list

In : map List.hd rows;
Out: - : int list = [1; 4]
In : map tl rows;
Out: - : int list list = [[2; 3]; [5; 6]]
In : transp rows;
Out: - : int list list = [[1; 4]; [2; 5]; [3; 6]]
```

Review of Matrix Multiplication

$$\begin{pmatrix} A_1 & \cdots & A_k \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ \vdots \\ B_k \end{pmatrix} = \begin{pmatrix} A_1 B_1 + \cdots + A_k B_k \end{pmatrix}$$

The right side is the *vector dot product* $\vec{A} \cdot \vec{B}$

Repeat for each *row* of A and *column* of B

Review of Matrix Multiplication

$$\begin{array}{ccc} \boldsymbol{A} & \boldsymbol{B} & \boldsymbol{A \times B} \end{array}$$

$$\begin{pmatrix} 2 & 0 \\ 3 & -1 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 2 \\ 4 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 4 \\ -1 & 1 & 6 \\ 4 & -1 & 0 \\ 5 & -1 & 2 \end{pmatrix}$$

For element (i,j) of A x B:
dot-product of row i and column j

Matrix Multiplication in OCaml

Dot product of two vectors—a **curried function**

```
let rec dotprod xs ys =
  match xs, ys with
  | [], [] -> 0.0
  | x::xs, y::ys -> x *. y +. dotprod xs ys
```

Q: What is the type of this function?

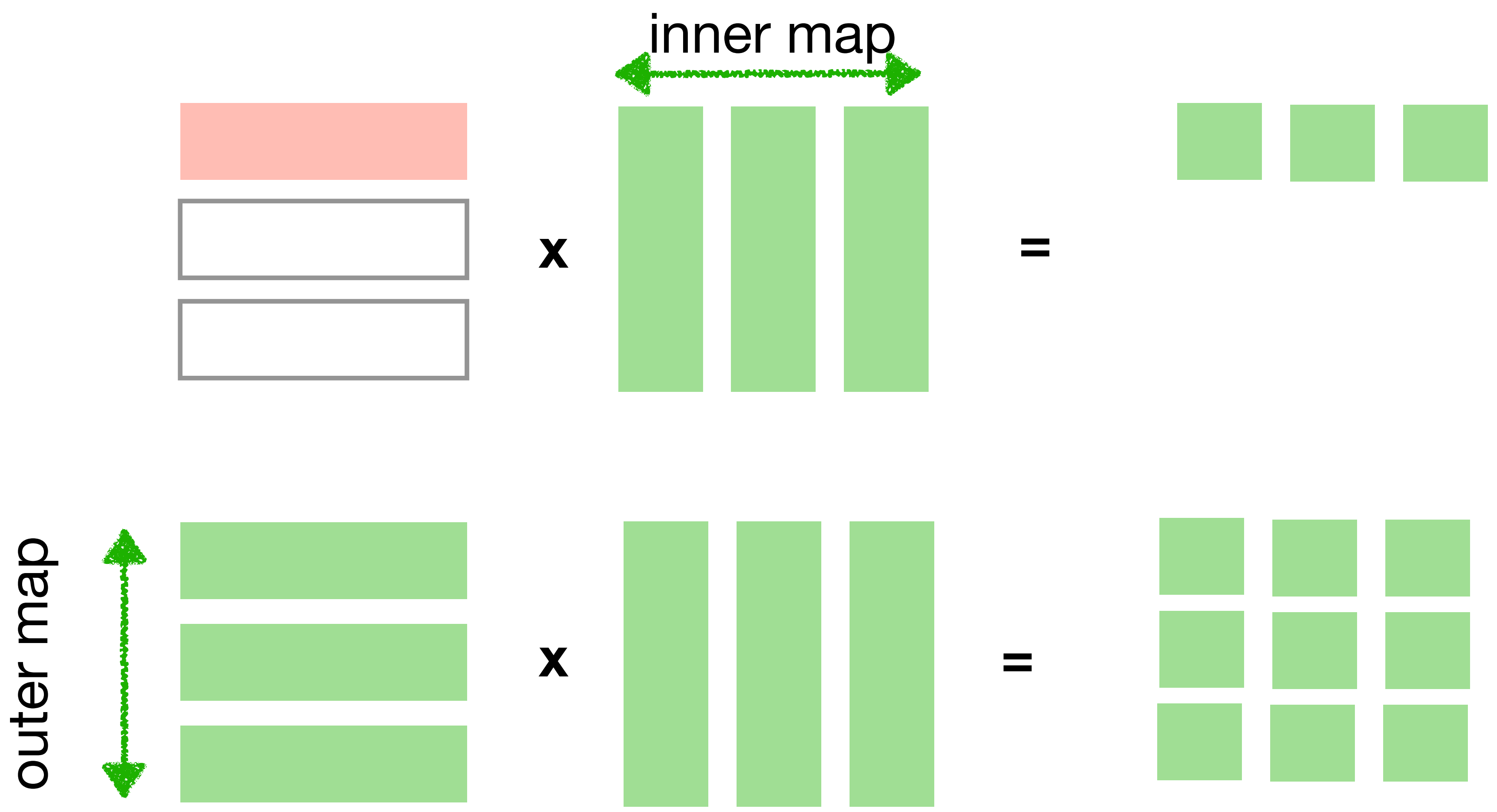
```
float list -> float list -> float
```

Matrix product

```
let matprod arows brows =
  let cols = transp brows in
  map (fun row -> map (dotprod row) cols) arows
```

Matrix Multiplication in OCaml

```
let rec matprod arows brows =
  let cols = transp brows in
  map (fun row -> map (dotprod row) cols) arows
```



List Functionals for Predicates

```
let rec exists p = function
| [] -> false
| x::xs -> (p x) || (exists p xs)
val exists : ('a -> bool) -> ('a list -> bool) = <fun>

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>
```

(A *predicate* is a *boolean-valued function.*)

List Functionals for Predicates

Dual to `exists`:

```
let rec all p = function
| [] -> true
| x::xs -> (p x) && all p xs

val all : ('a -> bool) -> 'a list -> bool = <fun>
```

Example:

```
> exists (fun x -> x mod 2 = 0) [1; 2; 3];;
- : bool = true

> filter (fun x -> x mod 2 = 0) [1; 2; 3];;
- : int list = [2]

> all (fun x -> x mod 2 = 0) [1; 2; 3];;
- : bool = false
```

Applications of the Predicate Functionals

```
let member y xs =
  exists (fun x -> x = y) xs;;

let inter xs ys =
  filter (fun x -> member x ys) xs;;
```

Testing whether two lists have no common elements

```
let disjoint xs ys =
  all (fun x -> all (fun y -> x <> y) ys) xs
```