

# **Higher Order Programming: Maps and Filters**

**Concepts of Programming Languages**  
**Lecture 9**

# Warm Up with Options (Syntax)

» Recall option types

```
type 'a option = None | Some of 'a
```

» Formal Syntax for Constructing Options:

```
<expr> ::= None | Some ( <expr> )
```

» Formal Syntax for Destructing Options:

```
<expr> ::= match <expr> with  
          | None -> <expr>  
          | Some ( <var> ) -> <expr>
```

# Typing Rules for Options

- » **None** is of type  $\tau$  option in context  $\Gamma$
- » If  $e$  is of type  $\tau$  in context  $\Gamma$ , then **Some( $e$ )** is of type  $\tau$  option in context  $\Gamma$

# Typing Rules for Options

$$\frac{}{\Gamma \vdash \text{None} : \tau \text{ option}} \text{ (none)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{Some}(e) : \tau \text{ option}} \text{ (some)}$$

» **None** is of type  $\tau$  option in context  $\Gamma$

» If  $e$  is of type  $\tau$  in context  $\Gamma$ , then **Some**( $e$ ) is of type  $\tau$  option in context  $\Gamma$

# Typing Rule for Option Match

If  $e$  is of type  $\tau'$  option in context  $\Gamma$ , and  
 $e_1$  is of type  $\tau$  in the same context  $\Gamma$ , and  
 $e_2$  is of type  $\tau$  in context  $\Gamma$  extended with  $x : \tau'$ , then  
**match**  $e$  **with** **| None**  $\rightarrow e_1$  **| Some**( $x$ )  $\rightarrow e_2$   
is of type  $\tau$  in context  $\Gamma$

# Typing Rule for Option Match

$$\frac{\Gamma \vdash e : \tau' \text{ option} \quad \Gamma \vdash e_1 : \tau \quad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \text{match } e \text{ with } | \text{None} \rightarrow e_1 | \text{Some}(x) \rightarrow e_2 : \tau} \text{ (opt-match)}$$

If  $e$  is of type  $\tau'$  option in context  $\Gamma$ , and  $e_1$  is of type  $\tau$  in the same context  $\Gamma$ , and  $e_2$  is of type  $\tau$  in context  $\Gamma$  extended with  $x : \tau'$ , then  $\text{match } e \text{ with } | \text{None} \rightarrow e_1 | \text{Some}(x) \rightarrow e_2$  is of type  $\tau$  in context  $\Gamma$

# Semantics Rules for Options

- » **None** evaluates to None
- » If  $e$  evaluates to  $v$ ,  
then **Some( $e$ )** evaluates to **Some( $v$ )**

# Semantics Rules for Options

$$\frac{}{\text{None} \Downarrow \text{None}} \text{ (none—eval)}$$

$$\frac{e \Downarrow v}{\text{Some}(e) \Downarrow \text{Some}(v)} \text{ (some—eval)}$$

- » **None** evaluates to None
- » If  $e$  evaluates to  $v$ ,  
then **Some**( $e$ ) evaluates to **Some**( $v$ )



# Semantic Rule 1 for Option Match

» If  $e$  evaluates to `None`, and  $e_1$  evaluates to  $v$ , then  
`match  $e$  with | None ->  $e_1$  |`  
`Some( $x$ ) ->  $e_2$`   
evaluates to  $v$

» If  $e$  evaluates to `Some( $v$ )`, and  $e_2$  with  $v$  substituted for  $x$  evaluates to  $v'$ , then  
`match  $e$  with | None ->  $e_1$  |`  
`Some( $x$ ) ->  $e_2$`   
evaluates to  $v'$

# Semantic Rule 1 for Option Match

$$\frac{e \Downarrow \text{None} \quad e_1 \Downarrow v}{\text{match } e \text{ with } | \text{None} \rightarrow e_1 | \text{Some}(x) \rightarrow e_2 \Downarrow v} \text{ (opt-eval-none)}$$

$$\frac{e \Downarrow \text{Some}(v) \quad e'_2 = [v/x]e_2 \quad e'_2 \Downarrow v'}{\text{match } e \text{ with } | \text{None} \rightarrow e_1 | \text{Some}(x) \rightarrow e_2 \Downarrow v'} \text{ (opt-eval-some)}$$

» If  $e$  evaluates to `None`, and  $e_1$  evaluates to  $v$ , then  
`match  $e$  with | None ->  $e_1$  |`  
`Some( $x$ ) ->  $e_2$`   
evaluates to  $v$

» If  $e$  evaluates to `Some( $v$ )`, and  $e_2$  with  $v$  substituted for  $x$  evaluates to  $v'$ , then  
`match  $e$  with | None ->  $e_1$  |`  
`Some( $x$ ) ->  $e_2$`   
evaluates to  $v'$

# Outline

- » Introduce the notion of **higher-order functions (HOFs)** as a way to write cleaner, more general code
- » Examine two common HOFs: **map** and **filter**

# Higher-Order Functions

# Higher-Order Programming

In OCaml, functions are **first-class values**

They can be:

1. returned by another function
2. given names with let-definitions
3. passed as arguments to another function

*Note.* Types are *not* first-class values

# Functions as Parameters

```
# let apply f x = f x;;  
val apply : ('a -> 'b) -> 'a -> 'b = <fun>  
# apply (fun x -> x+5) 10;;  
- : int = 15  
# let apply2 f x = f (f x);;  
val apply : ('a -> 'a) -> 'a -> 'a = <fun>  
# apply2 (fun x -> x+5) 10;;  
- : int = 20
```

This allows us to create new functions which are *parametrized* by old ones

# First-Order Function Types

`int -> string`

`t -> t`

`() -> bool`

`bool * bool -> bool`

# Second-Order Function Types

`(int -> string) -> (int -> string)`

`t -> (s -> t)`

`(() -> bool) -> bool`

`bool -> bool -> bool`



# Third-Order Functions

`(int -> string) -> (int -> string) -> (int -> string)`

`(t -> (s -> t)) -> t`

`((()) -> bool) -> bool`

`(bool -> bool -> bool) * bool -> bool`

# And so on...

```
1st: int
2nd: int -> int
3rd: (int -> int) -> int
4th: ((int -> int) -> int) -> int
5th: (((int -> int) -> int) -> int) -> int
6th: ((((int -> int) -> int) -> int) -> int) -> int
7th: ((((((int -> int) -> int) -> int) -> int) -> int) -> int) -> int) -> int
8th: (((((((int -> int) -> int) -> int) -> int) -> int) -> int) -> int) -> int) -> int
:
```

The **higher-order** part comes from the fact that we can do this *ad infinitum*

(In practice, we rarely use higher than third-order or fourth-order functions)

# **The Abstraction Principle**

# Motivation

- » Observe the main computation happening in a function; see what operations are needed
- » Can those operations become parameters? Yes! Because operations are like functions, and HOFs allow functions as parameters
- » We will see two examples: **map** and **filter**

# Observe the Pattern!

```
let rec fact n =  
  if n = 0 then 0  
  else n * fact (n - 1)
```

```
let rec sum n =  
  if n = 0 then 0  
  else n + sum (n - 1)
```

*Can we abstract the core functionality?*

demo  
(accumulate)

# Accumulation Pattern

```
let rec accum f n start =  
  let rec go n =  
    match n with  
    | 0 -> start  
    | n -> f n (go (n - 1))  
  in go n
```

In order to generalize this function, we need to be able to take the *operation as a parameter*

Now we have a single function which we can *reuse* elsewhere

# Practice Problem

*Implement the function*

**val sum\_squares : int -> int**

*so that sum\_squares n is the sum  $1^2 + 2^2 + \dots + n^2$*

**val sum\_cubes : int -> int**

*so that sum\_cubes n is the sum  $1^3 + 2^3 + \dots + n^3$*

*Write a single function that can be used to implement both*





# Map

# Observe the Pattern!

- » Each element is being modified
- » Modification function is the same for all elements
- » **map** is used to apply a function to every element in a list (or other structure)

```
int list -> int list
let rec inc l =
  match l with
  | [] -> []
  | h::t -> (h+1)::(inc t)

int list -> int list
let rec dec l =
  match l with
  | [] -> []
  | h::t -> (h-1)::(dec t)

int list -> int list
let rec double l =
  match l with
  | [] -> []
  | h::t -> (2*h)::(double t)
```

# Definition of Map

```
let rec map f l =  
  match l with  
  | [] -> []  
  | x :: xs -> f x :: map f xs
```

*Is this tail recursive?*

- » *If the list is empty there is nothing to do*
- » *If the list is nonempty, we apply f to its first element, and recurse*

# Tail-Recursive Map

```
let rec map_tr f l =  
  let rec go l acc =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go xs (f x :: acc)  
  in go l []
```

For a tail-recursive version we can build the list in reverse in acc and then *reverse it at the end*

This may seem inefficient, but it's just a *constant factor* slower

# They are so easy now!

```
int list -> int list
let inc l = map (fun x -> x + 1) l
```

```
int list -> int list
let dec l = map (fun x -> x - 1) l
```

```
int list -> int list
let double l = map (fun x -> 2 * x) l
```

```
int list -> int list
let rec inc l =
  match l with
  | [] -> []
  | h::t -> (h+1)::(inc t)
```

```
int list -> int list
let rec dec l =
  match l with
  | [] -> []
  | h::t -> (h-1)::(dec t)
```

```
int list -> int list
let rec double l =
  match l with
  | [] -> []
  | h::t -> (2*h)::(double t)
```

# demo

(int/float lists)

# Practice Problem

*Implement the function*

***val pointwise\_max : ('a -> 'b) -> ('a -> 'b)  
-> 'a list -> 'b list***

*so that pointwise\_max f g l is l but with f or g applied to each element, whichever gives the larger value*



**Filter**



# Observe the Pattern!

```
int list -> int list
let rec evens l =
  match l with
  | [] -> []
  | x::xs -> (if x mod 2 = 0 then [x] else []) @ evens xs
```

```
int list -> int list
let rec pos l =
  match l with
  | [] -> []
  | x::xs -> (if x >= 0 then [x] else []) @ pos xs
```

**filter** is used to grab all elements in a list which *satisfy a given property*

# Predicates

**Definition:** A Boolean predicate on 'a' is a function of type 'a -> bool'

A predicate is a function which defines a *property*

Examples:

```
let even n = n mod 2 = 0
```

```
let even_length l = even (List.length l)
```

# Definition of Filter

```
let rec filter p l =  
  match l with  
  | [] -> []  
  | x :: xs ->  
    (if p x then [x] else []) @ filter p xs
```

***Is this tail recursive?***

- » *If the list is empty there is nothing to do*
- » *If the first element satisfies our predicate we keep it and recurse*
- » *Otherwise, we drop it and recurse*

# Tail-Recursive Definition of Filter

```
let filter_tr p =  
  let rec go acc l =  
    match l with  
    | [] -> List.rev acc  
    | x :: xs -> go ((if p x then [x] else []) @ acc) xs  
  in go []
```

As with map, we have to reverse the output before returning it

# This is so easy now!

```
int list -> int list
```

```
let evens l = filter (fun x -> x mod 2 = 0) l
```

```
int list -> int list
```

```
let pos l = filter (fun x -> x >= 0) l
```

**filter** just needs a filtering function as parameter

# Understanding Check

```
let h p q = filter (fun i -> p i && q i)
```

*What does the above function do?*

# demo

(filter function on functions)

# Summary

**Higher-order function** allow for better **abstraction** because we can **parameterize** functions by other functions

**List.map** and **List.filter** are very common patterns which can be used to write clean and simple code