Higher Order Programming: Maps and Filters

Concepts of Programming Languages
Lecture 7

CAS CS 320

Outline

» Introduce the notion of **higher-order functions** as a way to write cleaner, more general code

» Examine two common HOFs: **map** and **filter**

Practice Problem

{ x : int, y : int } ⊢ x + if x = y then x else y : int

Give a derivation of the above typing judgment

Solution

$$
\dfrac{
\dfrac{}{\{x:\mathrm{int}, y:\mathrm{int}\} \vdash x : \mathrm{int}}\,(\mathrm{var})
\qquad
\dfrac{
\dfrac{
\dfrac{}{\{x:\mathrm{int}, y:\mathrm{int}\} \vdash x : \mathrm{int}}\,(\mathrm{var})
\quad
\dfrac{}{\{x:\mathrm{int}, y:\mathrm{int}\} \vdash y : \mathrm{int}}\,(\mathrm{var})
}{\{x:\mathrm{int}, y:\mathrm{int}\} \vdash x = y : \mathrm{bool}}\,(\mathrm{eq})
\quad
\dfrac{}{\{x:\mathrm{int}, y:\mathrm{int}\} \vdash x : \mathrm{int}}\,(\mathrm{var})
\quad
\dfrac{}{\{x:\mathrm{int}, y:\mathrm{int}\} \vdash y : \mathrm{int}}\,(\mathrm{var})
}{\{x:\mathrm{int}, y:\mathrm{int}\} \vdash \text{if } x = y \text{ then } x \text{ else } y : \mathrm{int}}\,(\mathrm{if})
}{\underbrace{\{\; x : \mathtt{int},\; y : \mathtt{int} \;\}}_{\Gamma} \vdash \underbrace{x}_{e_1} + \underbrace{(\mathtt{if}\ x = y\ \mathtt{then}\ x\ \mathtt{else}\ y)}_{e_2} : \mathtt{int}}\,(\mathrm{intAdd})
$$

Higher-Order Functions

Higher-Order Programming

Higher-Order Programming

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

Higher-Order Programming

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

They can be:

Higher-Order Programming

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

They can be:

1. returned by another function

Higher-Order Programming

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

They can be:

1. returned by another function
2. given names with let-definitions

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

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

Aside: Robin Popplestone

"He started a PhD at Manchester University before moving to Leeds University. His project was to develop a program for automated theorem proving, but he got caught up in **using the university computer to design a boat.** He built the boat and set sail for the University of Edinburgh, where he had been offered a research position. A storm hit while crossing the North Sea, and **the boat sank.** A widely believed story about Popplestone was that he never completed his PhD in mathematics because he **lost his thesis manuscript in the boat,** although Popplestone refused to corroborate this."

https://en.wikipedia.org/wiki/Robin_Popplestone

Functions as Return Values

```
# let f x y = x + y;;
val f : int -> int -> int = <fun>
# f 2;;
- : int -> int = <fun>
```

This isn't that interesting in OCaml...

Functions in OCaml are **Curried**, so multi-argument functions return functions already

Functions as Return Values

```
# let f x y = x + y;;
val f : int -> (int -> int) = <fun>
# f 2;;
- : int -> int = <fun>
```

This isn't that interesting in OCaml...

Functions in OCaml are **Curried**, so multi-argument functions return functions already

Functions as Named Values

```
let f x y = x + y
```

is shorthand for...

```
let f = fun x -> fun y -> x + y
```

This also isn't that interesting in OCaml...

When we **let-define** *any* function, we're giving a anonymous function value a name

Functions as Named Values

```
let f x y = x + y
```

is shorthand for...

anonymous function

```
let f = fun x -> fun y -> x + y
```

This also isn't that interesting in OCaml...

When we **let-define** *any* function, we're giving a anonymous function value a name

Functions as Parameters

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

This is *very* interesting in OCaml...

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

Functions as Parameters

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

note the type

This is *very* interesting in OCaml...

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

Higher-Order Functions Elsewhere

fun $f \to \dfrac{f(x)}{dx}$ **e.g.** $x^2 \mapsto 2x$

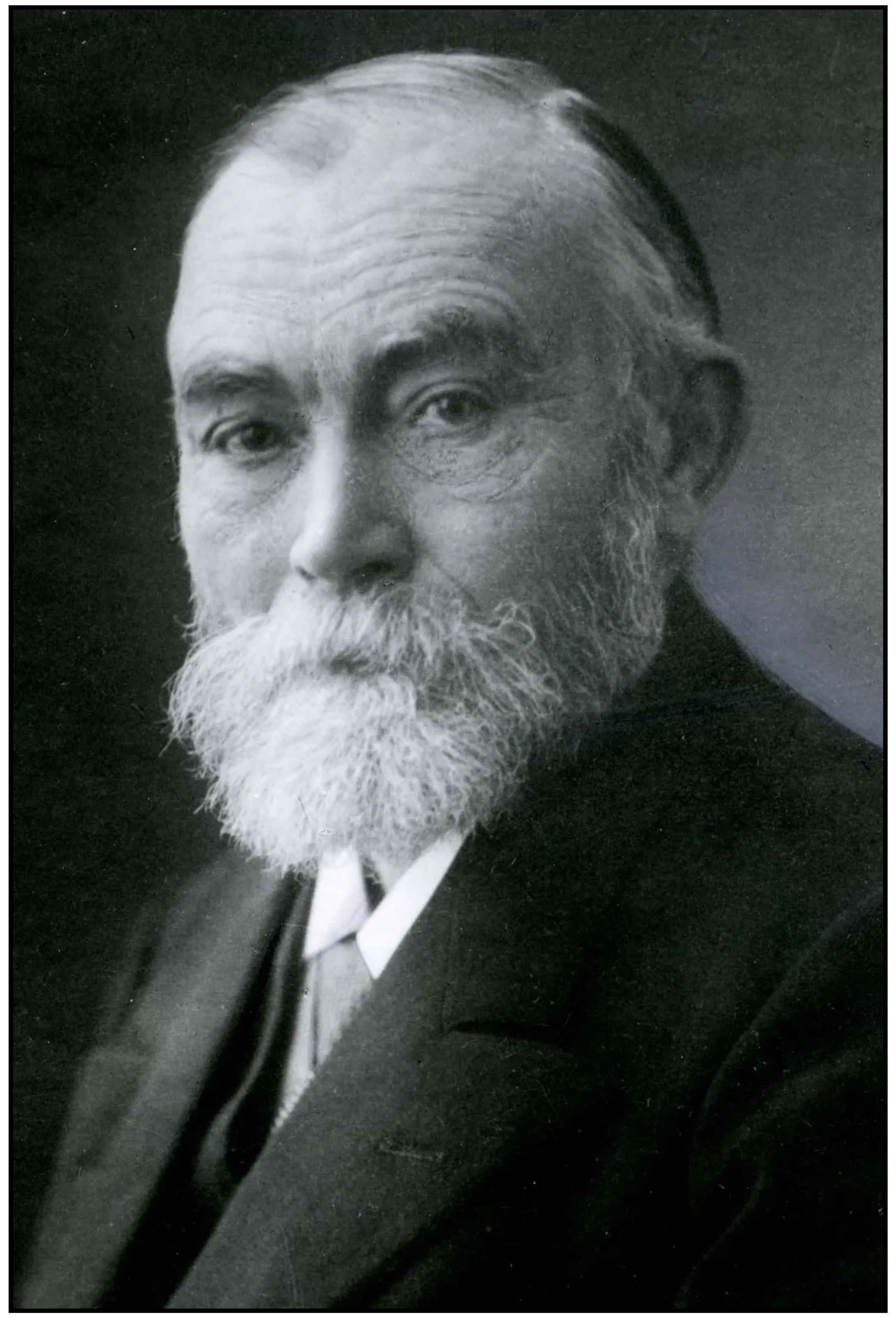
`We might think of the type of an` **`derivative`** `as`

$$(\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$$

`because it takes one function and produces a new function`

Aside: What does "Higher-Order" Mean?

"Like things and functions are different, so are functions whose **arguments are functions** *radically different* from functions whose **arguments must be things**. I call the latter functions of first order, the former functions of second order."



Gottlob Frege

https://softwareengineering.stackexchange.com/questions/186035/who-first-coined-the-term-higher-order-function-and-or-first-class-citizen

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 -> 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
8th: ((((((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

Motivation

One of the three virtues of a great programmer is *laziness*

Motivation

One of the three virtues of a great programmer is *laziness*

The **abstraction principle** helps use be lazy

Motivation

One of the three virtues of a great programmer is *laziness*

The **abstraction principle** helps use be lazy

When we write general programs, we *avoid rewriting programs* we've (pretty much) written before

Simple Example

```
                         'a                'a
let rec reverse (l : int list) : int list =
  match l with
  | [] -> []
  | x :: xs -> reverse xs @ [x]
```

Remember that **polymorphism** allows us to write general functions by being *agnostic* about types

*It doesn't matter if we're reversing an **int list** of **string list** or an **int list list**...*

Simple Example

```
let rec fact n =
  match n with
  | 0 -> 1
  | n -> n * fact (n - 1)
```

int -> int -> int

```
let rec sum n =
  match n with
  | 0 -> 0
  | n -> n + sum (n - 1)
```

int -> int -> int

Some functions cannot be polymorphic

But can we still abstract the core functionality?

Simple Example

```
let rec fact n =
  match n with
  | 0 -> 1
  | n -> n * fact (n - 1)

let rec sum n =
  match n with
  | 0 -> 0
  | n -> n + sum (n - 1)
```

η- red.

eta- red.

Some functions cannot be polymorphic

But can we still abstract the core functionality?

demo

(accumulate)

Simple Example

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

Simple Example

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

Simple Example

```
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*

Simple Example

```
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

Another Example

```
let rec insert (x : 'a) (l : 'a list) : 'a list =
  match l with
  | [] -> [x]
  | y :: ys -> if x <= y then x :: y :: ys else y :: insert x ys

let rec sort (l : 'a list) : 'a list =
  match l with
  | [] -> []
  | x :: xs -> insert x (sort xs)
```

Sorting *is* polymorphic

But what if we want to sort in *reverse order*, or *only on a part of the data*?

demo

(sorting)

The Abstraction Principle

The Abstraction Principle

The abstraction principle comes from MacLennan's
Functional Programming: Theory and Practice

The Abstraction Principle

The abstraction principle comes from MacLennan's **Functional Programming: Theory and Practice**

» Abstract out <u>core functionality</u>

The Abstraction Principle

The abstraction principle comes from MacLennan's **Functional Programming: Theory and Practice**

- Abstract out core functionality
- Use higher-order functions to parametrize by functionality specific to the problem

The Abstraction Principle

The abstraction principle comes from MacLennan's **Functional Programming: Theory and Practice**

» Abstract out core functionality

» Use higher-order functions to parametrize by functionality specific to the problem

» (Try to understand the algebra of programming)

Understanding Check

Implement the function

`val negatives : int list -> int list`

so that **`negatives l`** is the list negative numbers appearing in **`l`.**
Also implement the function

`val gets : 'a -> ('a * 'b) list -> 'b list`

so that **`gets key l`** is the list of values **`v`** such that **`(key, v)`** is a member of **`l`**

Write a single function that can be used to implement both

Map

Example

```
type user = {
  name : string ;
  id : int ;
}

let capitalize = ...

let fix_usernames (us : user list)  =
  List.map (fun u -> { u with name = capitalize u.name }) us
```

`map` is used to apply a function to every element in a list (or other structure)

Definition of Map

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

Definition of Map

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

» *If the list is empty there is nothing to do*

Definition of Map

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

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

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

Tail-Recursive Map

```
let rec map_t 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*

Tail-Recursive Map

```
let rec map_t 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 its just a *constant factor* slower

Additional Notes

Additional Notes

The text mentions two additional things about map:

Additional Notes

The text mentions two additional things about map:

- There is a function **rev_map**, which is tail-recursive and does give the output in reverse order

Additional Notes

The text mentions two additional things about map:

- There is a function **rev_map**, which is tail-recursive and does give the output in reverse order
- map is defined somewhat differently to account for side-effects

Additional Notes

The text mentions two additional things about map:

» There is a function **rev_map**, which is tail-recursive and does give the output in reverse order

» map is defined somewhat differently to account for side-effects

We won't dwell on these for now, but it may be worth reading about

demo

(normalize)

Understanding Check

Implement then function

```
val pointwise_max : ('a -> int) -> ('a -> int)
                    -> 'a list -> 'a 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

Example

```
type user = {
  name : string ;
  id : int ;
  num_likes : int ;
}

let popular (us : user list) (cap : int) =
  List.filter (fun u -> u.num_likes > cap) us
```

filter is used to do 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
```

Definition of Filter

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

» *If the list is empty there is nothing to do*

Definition of Filter

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

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

Definition of Filter

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

» *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*

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_tail 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 demo

(primes)

Understanding Check

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

What does the above function do?

Summary

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

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