

INF231:
Functional Algorithmic and Programming
Lecture 6: Polymorphism, Higher-order, and Currying

Academic Year 2020 - 2021

$f(x)$



Outline

Polymorphism

Higher-Order

Currying

Motivating polymorphism on examples

Limitations of Functions

About the *identity* function:

- ▶ Identity on `int`:

```
let id (x:int):int = x           val id:int → int = <fun>
```

- ▶ Identity on `float`:

```
let id (x:float):float = x      val id:float → float = <fun>
```

- ▶ Identity on `char`

```
let id (x:char):char = x       val id:char → char = <fun>
```

Disadvantages:

- ▶ **1 function per type** needing the identity function
- ▶ Unique/Different names needed if these functions should “live” together

Motivating polymorphism on examples

Limitations of Functions on list

Compute the length of a list:

▶ of int:

```
let rec length_int (l: int list):int=  
  match l with  
  | [] → 0  
  | _::l → 1+ length_int l
```

▶ on char:

```
let rec length_char (l: char list):int=  
  match l with  
  | [] → 0  
  | _::l → 1+ length_char l
```

▶ ...

Remark The body of these functions is not specific to char nor int



→ we need lists that are *not bound to a type*

Motivating polymorphism on examples

Limitations of (current) lists

Several sorts of lists, à la Lisp:

- ▶ `type listofint = Nil | Cons int * listofint`
and then `Cons (2, Cons (9,Nil))`
- ▶ `type listofchar = Nil | Cons char * listofchar`
and then `Cons ('t', Cons ('v',Nil))`

Several sorts of lists, even with OCaml shorter notations:

- ▶ list of int: `[1;2]` (`=1::2::[]`) of type `int list`
- ▶ list of char: `['e';'n']` (`= 'e'::'n'::[]`) of type `char list`
- ▶ list of string: `["toto";"titi"]` (`= "toto"::"titi"::[]`) of type `string list`

Back on the examples - introducing polymorphism

Let's come back on the (various) identity functions

What if we omit type?

```
let id x = x                                val id : 'a → 'a = <fun>
```

→ type inference: OCaml computes the most general type

→ *polymorphic* identity: the id on any type (α or 'a)

“id is a *polymorphic* function” that can be applied to any value

We can specifically indicate that the function can take any type:

```
let id (x : 'a) : 'a = x
equivalently let id (x : 'b) : 'b = x
equivalently let id (x : toto) : toto = x
...
...
↪ the type returned by OCaml is 'a → 'a
(equivalently  $\alpha \rightarrow \alpha$ )
```

DEMO: Polymorphic identity

Polymorphic lists

We can define lists that are parameterized by some type (à la Lisp)

```
type 't llist = Nil | Cons of 't * 't llist
```

't is a **type parameter**

OCaml pre-defined lists are already parameterized by some type:

- ▶ type of [] is 'a list (equivalently α list)
- ▶ type of :: is 'a → 'a list → 'a list
(equivalently $\alpha \rightarrow \alpha$ list $\rightarrow \alpha$ list)

Remark Still, the elements should have the same type



Example

- ▶ Cons (2, Cons (3, Cons (4, Nil)))
- ▶ Cons ('r', Cons ('d', Cons ('w', Nil)))
- ▶ Cons ((fun x → x), Cons ((fun x → 3*x+2), Nil))

DEMO: Polymorphic lists

Polymorphic functions on Polymorphic lists

Let's practice

Example (Length of a list)

À la Lisp:

```
let rec length (l:'a list):int
  =
  match l with
  | Nil → 0
  | Cons (_, l) → 1 + length l
```

OCaml pre-defined lists:

```
let rec length (l:'a list):int
  =
  match l with
  | [] → 0
  | _::l → 1 + length l
```

Exercise: implement some functions on polymorphic lists

- ▶ `isEmpty`: returns `true` if the argument list is empty
- ▶ `append`: appends two lists together
- ▶ `reverse`: reverse the elements of a list
- ▶ `separate`: inserts a separator between two elements of a list

Note: be careful with the types

Outline

Polymorphism

Higher-Order

Currying

Higher-order

Some motivation

Consider two simple functions returning the maximum of two integers:

```
let max2_v1 (a:int) (b:int):int =  
  if a >= b then a  
  else b
```

```
let max2_v2 (a:int) (b:int):int =  
  if a <= b then b  
  else a
```

Several questions:

- ▶ How to test whether those functions are correct?
- ▶ How to test whether those functions return the same values for the same input values?

DEMO: Higher-order can provide elegant testing solutions

Introduction to higher-order

In OCaml and functional programming, functions is the basic tool:

- ▶ to “slice” a program into smaller pieces
- ▶ to produce results

Functions are first-class citizens: they are values (e.g., used in lists,...

A function can also be *a parameter or a result of a function*

Example (Returning an affine function)

```
let affine a b = (fun x → a* x + b)
```

Several benefits

Higher-order functions

Some vocabulary

Definition (Higher-Order language)

A programming language is a higher-order language if it allows to pass functions as parameters **and** functions can return a function as a result

Remark The C programming language allows to pass functions as parameters but does not allow to return a function as a result



Definition (Higher-order function)

A function is said to be a **higher-order function** or a **functional**, if it does at least one of the two things:

- ▶ take at least one function as a parameter
- ▶ return a function as a result

Remark Non higher-order functions are said to be *first-order* functions



Higher-order functions

Benefits and What you should learn

Conciseness

Some form of expressiveness

At the end of the day, you should know:

- ▶ that higher-functions exist
- ▶ the associated "vocabulary"
- ▶ **know how and when to use it**

We will demonstrate and experiment those features through examples. . .

A tour of some higher-order functions

Numerical functions

Example (Slope of a function in 0)

Let f be a function defined in 0 (with real values):

$$\frac{f(h) - f(0)}{h} \quad (\text{with } h \text{ small})$$

DEMO: Slope in 0

Example (Derivating a (derivable) function f)

We approximate $f'(x)$ (value of the derivative function in x) by:

$$\frac{f(x + h) - f(x)}{h} \quad (\text{with } h \text{ small})$$

DEMO: Derivative

A tour of some higher-order functions

Numerical functions

Reminders:

- ▶ A *zero* of a function f is an x s.t. $f(x) = 0$
- ▶ Theorem of *intermediate values*:
Let f be a continuous function, a and b two real numbers, if $f(a)$ and $f(b)$ are of opposite signs, then there is a zero in the interval $[a, b]$
- ▶ \sqrt{a} is the positive zero of the function $x \mapsto x^2 - a$
- ▶ $\forall a \geq 0 : 0 \leq \sqrt{a} \leq \frac{1+a}{2}$

Exercise: zero of a continuous function using dichotomy

- ▶ Define a function `sign` indicating whether a real is positive or not
- ▶ Deduce a function `zero` that returns the zero of a function, up to some given epsilon, given two reals s.t. there is a zero between those reals
- ▶ Deduce a function to approximate the square root of a float

A tour of some higher-order functions

Applying twice a function

Consider the two functions `double` and `square`:

- ▶ `let double (x:int):int = 2*x`
- ▶ `let square (x:int):int = x*x`

How can we define `quad` and `power4` reusing the previous function?

- ▶ `let quad (x:int):int = double (double x)`
- ▶ `let square (x:int):int = square (square x)`

Can we generalize?... Yes, we can:

```
let applyTwice (f:int → int) (x:int):int = f (f x)
```

- ▶ `let quad (x:int):int = applyTwice double x`
- ▶ `let power4 (x:int):int = applyTwice square x`

or using anonymous functions:

- ▶ `let quad (x:int):int = applyTwice (fun (x:int) → 2* x) x`
- ▶ `let power4 (x:int):int = applyTwice (fun (x:int) → x * x) x`

A tour of some higher-order functions

Composing functions

Function composition:

$$\begin{aligned} f &: C \longrightarrow D \\ g &: A \longrightarrow B \\ g \circ f &: C \longrightarrow B \quad \text{if } D \subseteq A \end{aligned}$$

Let us simplify and take $D = A$, hence $g \circ f : C \xrightarrow{f} A \xrightarrow{g} B$

Exercise: Defining function composition in OCaml

- ▶ Specify the function `compose` that composes two functions (beware of types)
- ▶ Implement the function `compose`

A tour of some higher-order functions

n-th term of a series and generalized composition

Consider a series defined as follows:

$$\begin{aligned}u_0 &= a \\ u_n &= f(u_{n-1}), n \geq 1\end{aligned}$$

The *n*-th term u_n is $f(u_{n-1}) = f(f(u_{n-2})) = f(f(f(\dots(u_0)\dots)))$

Exercise: *n*-th term of a series

Define a function `nthTerm` that computes the *n*-th term of a series defined as above using a function *f* and some *n*

Exercise: *n*-th iteration of a function

Define a function `iterate` that computes the function which is the *n*-th composition of a function, given some *n*

A tour of some higher-order functions

Generalizing the sum of the n first integers

Sum of n first integers:

$$1 + 2 + \dots + (n - 1) + n = (1 + 2 + \dots + (n - 1)) + n$$

Implemented as:

```
let rec sum_integers (n:int) =  
  if n=0 then 0 else sum_integers (n-1) + n
```

The sum of squares is similarly:

$$1^2 + 2^2 + \dots + (n - 1)^2 + n^2 = (1^2 + 2^2 + \dots + (n - 1)^2) + n^2$$

Implemented as:

```
let rec sum_squares (n:int) =  
  if n=0 then 0 else sum_squares (n-1) + (n*n)
```

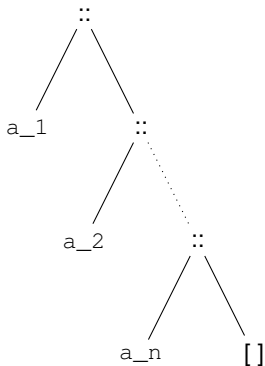
Sum of the integers through a function - generalization

- ▶ Define a function `sigma` that computes the sum of the images through some function for the first n integers
- ▶ Give an alternative implementation of `sum_integers` and `sum_squares` using `sigma`

A tour of some higher-order functions

Lists: applying a function to all elements in a list - preliminary

Another representation of the list $l = [a_1; a_2; \dots; a_n]$:



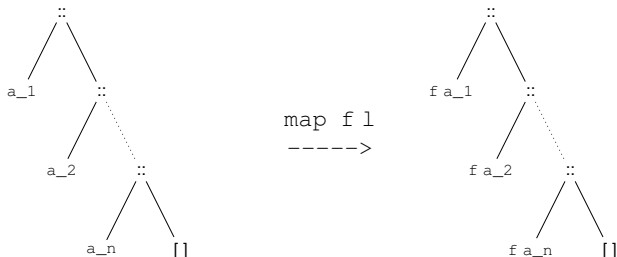
Graphic representation from Pierre Wiels and Xavier Leroy

A tour of some higher-order functions

Lists: applying a function on all elements on a list - function `map`

Given:

- ▶ a list of type 'a list
- ▶ a function of type 'a → 'b



Remark

- ▶ Application of `f` does not depend on the position of the element
- ▶ `map` returns a list
- ▶ `map` can change the type of the list

Typing

If `l` is of type `t1 list` and `f` is of type `t1 → t2`
then `map f l` is of type `t2 list`



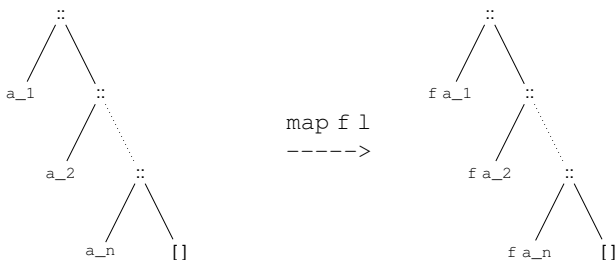
A tour of some higher-order functions

Lists: applying a function on all elements on a list - function `map`

Exercise: function `map`

Define a function `map` such that:

- ▶ given a list and a function f on the elements of that list,
- ▶ returns the list where f has been applied to all elements of that list



A tour of some higher-order functions

Lists: applying a function on all elements on a list - function `map`

Example (Vectorize)

► Specification:

- Profile: `vectorize: Seq(Elt) → Vec(Seq(Elt))`, where `Vec` is the set of lists of one element

► Semantics:

`vectorize [e1;...;en] = [[e1] ; ... ; [en]]`

► Implementation:

```
let vectorize = my_map (fun e → [e])
```

Example (Concatenate to each)

► Specification:

- Profile: `Seq(Elt) * Séq(Seq(Elt)) → Seq(Vec(Elt))`

► Semantics:

`concatenate_to_each (l, [v1; ...; vn] = [l@v1 ; ... ; l@vn]`

► Implementation:

```
let concatenate_to_each  
  = fun (l,seqv) → my_map (fun x → l@x) seqv
```

A tour of some higher-order functions

Lists: applying a function on all elements on a list - function `map`

Exercise: using the function `map` for converting lists

Define the following functions:

- ▶ `toSquare`: raises all elements of a list of `int` to their square
- ▶ `toAscii`: returns the ASCII code of a list of `char`
- ▶ `toUpperCase`: returns a list of `char` where all elements have been put to uppercase

Exercise: Powerset

Define the function `powerset` that computes the set of subsets of a set represented by a list

A tour of some higher-order functions

Lists: iterating a function on all elements on a list - function `fold_right` - some intuition first

Example (Sum of the elements of a list)

```
let rec sum l =  
  match l with  
  [] → 0  
  | elt::remainder → elt + (sum remainder)
```

Example (Product of the elements of a list)

```
let rec product l =  
  match l with  
  [] → 1  
  | elt::remainder → elt * (product remainder)
```

Example (Paste the string of a list)

```
let rec concatenate l =  
  match l with  
  [] → " "  
  | elt::remainder → elt ^ (concatenate remainder)
```

Remark Notice that the only elements that change are:

- ▶ the “base case”, i.e., what the function should return on the empty list
- ▶ “how we combine the current element with the result of the recursive call



A tour of some higher-order functions

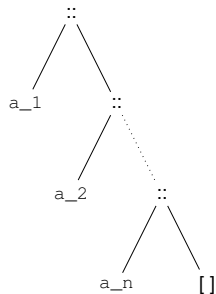
Lists: iterating a function on all elements on a list - function `fold_right`

If we place the operator in prefix position, we have:

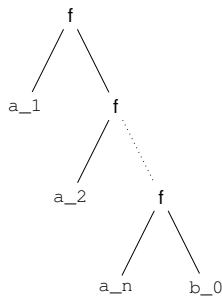
- ▶ `sum [a1;a2;...;an] = + a1 (+ a2 (... (+ an 0)...))`
- ▶ `product [a1;a2;...;an] = * a1 (* a2 (... (* an 0)...))`
- ▶ `concatenate [a1;a2;...;an] = ^ a1 (^ a2 (... (^ an 0)...))`

More generally, given:

- ▶ `f` of type `'a → 'b → 'b`,
- ▶ `l` of type `'a list`, and
- ▶ some initial value `b` of type `'b`



`fold_right f l b`
----->



→ result is of type `'a`

A tour of some higher-order functions

Lists: iterating a function on all elements on a list - function `fold_right`

Exercise: writing `fold_right`

Given

- ▶ `f` of type `'a → 'b → 'b`, and
- ▶ `l = [a1;...;an]` of type `'a list`,

define a function `fold_right` s.t.

$$\text{fold_right } f \text{ } [a1;...;an] \text{ } b = f (a1 (... f (an b)))$$

Exercise: using `fold_right`

- ▶ Re-write the previously defined functions, `sum`, `product`, `concatenate` using `fold_right`
- ▶ Define a function that determines whether the number of elements of a list is a multiple of 3 without using the function returning the length of a list

A tour of some higher-order functions

A small case-study with `fold_right`

Exercise: tasting testing

The purpose is to write a test suite function

We have seen examples of test cases

A test suite is a series of test cases s.t.:

- ▶ each test case is applied in order
- ▶ for a test suite to succeed, all its test cases must succeed

Questions:

- ▶ Define a function `test_suite` that checks whether two functions `f` and `g` returns the same values on a list of inputs values. Each element of the list is an input to the two functions.
- ▶ Here are two simple functions:
 - ▶ `let plus1 = fun x → x+1`
 - ▶ `let plus1dummy = fun x → if (x mod 2 = 0) then x -2 + 3
else 2*x`

Find 2 lists of inputs, so that the application of the function `test_suite`

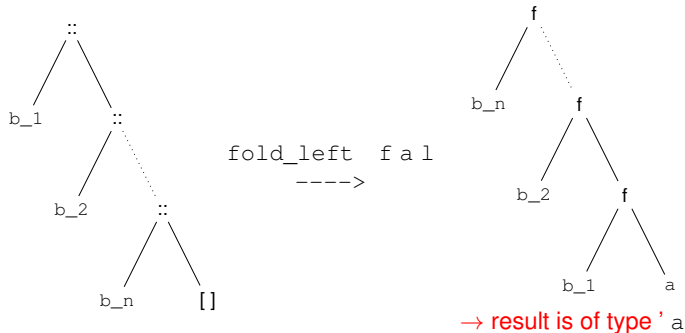
1. finds the bug
2. does not find the bug

A tour of some higher-order functions

Lists: iterating a function on all elements on a list - function `fold_left`

More generally, given:

- ▶ f of type $'a \rightarrow 'b \rightarrow 'a$,
- ▶ l of type $'b$ list, and
- ▶ some initial value a of type $'a$:



A tour of some higher-order functions

Lists: some function parameterized by a predicate

A predicate is a function that returns a Boolean

Recall the function that removes not even integers from a list of integers:

```
let rec remove_odd (l:int list) =  
  match l with  
  | [] → []  
  | elt::remainder →  
    if elt mod 2 = 0  
    then elt::(remove_odd remainder)  
    else (remove_odd remainder)
```

A tour of some higher-order functions

Lists: some function parameterized by a predicate

Exercise: Filtering according to a predicate

Define a function `filter` that filters the elements of a list according to some given predicate `p`

Exercise: Checking a predicate on the elements of a list

- ▶ Define a function `forall` that checks whether *all* the elements of a list satisfy a given predicate `p`
- ▶ Define a function `exists` that checks whether *at least one* element of a list satisfy a given predicate `p`

A tour of some higher-order functions

Some more exercises

Exercise: back to testing

- ▶ Redefine the function `test_suite` using the function `forall`

Exercise: Map with fold

- ▶ Redefine `map` using `fold_left`
- ▶ Redefine `map` using `fold_right`

Exercise: minimum and maximum with one line of code

Define the functions `minimum` and `maximum` of a list using `fold_left` and `fold_right`. The function can be written with one line of code

Outline

Polymorphism

Higher-Order

Currying

About Currying

A function with n parameter x_1, \dots, x_n is actually a function that takes x_1 as a parameter and returns a function that takes x_2, \dots, x_n as parameters

The application

$$f \ x_1 \ x_2 \ \dots \ x_n$$

is actually a series of applications

$$f \ (\dots \ (f \ x_1) \ x_2) \ \dots \ x_n$$

Definition: Partial application

Applying a function with n parameters with (strictly) less than n parameters
The result of a partial application remains a function

Typing:

If

- ▶ f is of type $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$, and
- ▶ x_i is of type t_i for $i \in [1, j] \subseteq [1, n]$

Then $f \ x_1 \ x_2 \ \dots \ x_j$ is of type $t_{(j+1)} \rightarrow \dots \rightarrow t_n \rightarrow t$

About Currying

Some example

Example (Apply twice)

Back to the function `applyTwice`:

```
let applyTwice (f:int → int) (x:int):int
  = f (f x)
```

Applying `applyTwice` with only one argument:

```
applyTwice (fun x → x +4)
```

is equal to the function

```
fun x → x + 8
```

DEMO: `applyTwice` and its testing

Currying has some advantages

Suppose we want a function taking $a \in A$ and $b \in B$ and returning $c \in C$

<u>Without currying:</u>	:	<u>With currying:</u>
$f: tA * tB \rightarrow tC$:	$f: tA \rightarrow tB \rightarrow tC$
f takes 1 argument: a pair	:	f takes 2 arguments
$f(a,b)$ is of type tC	:	$f a b$ is of type tC
	:	$f a$ is of type $tB \rightarrow tC$

DEMO: 2 definitions of integer addition & the predefined (+) in OCaml

Lessons learned

- ▶ Currying allows some *flexibility*
- ▶ Allows to *specialize* functions

Remark When applying curried functions, it can be harder to detect that we have forgot a parameter □

Conclusion / Summary

Polymorphism

- ▶ general types
- ▶ "type parameterization"

Higher-Order

- ▶ "taking a function as a parameter or returning a function"
- ▶ improve conciseness, expressiveness, quality, . . .

Currying

- ▶ partial application of a function
- ▶ function specialization
- ▶ define your function so it can be curried