GRENOBLE 1
UNIVERSITE JOSEPH FOURIER
SCIENCES. TECHNOLOGIE. SANTÉ

INF231: Functional Algorithmic and Programming

Lecture 6: Polymorphism, Higher-order, and Currying

Academic Year 2019 - 2020

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

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'] (=' b ':: ' 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 $\alpha \to \alpha$)

DEMO: Polymorphic identity

Polymorphic lists

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

$$\texttt{type 't llist = Nil | Cons of 't * 't llist}$$

'`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

Example (Length of a list)

À la Lisp:

```
let rec length (l:'a llist):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

Why it is useful to have 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 power4 (x:int):int = square (square x)`

The same: apply a given function twice to a value. By passing in the function to another function `twice` as an argument, we can abstract this fun:

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

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

or using anonymous functions:

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

Example (Slope of a function in 0)

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

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

DEMO: Slope in 0

Example (Derivating a (derivable) function f)

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

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

DEMO: Derivative

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

Function composition:

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

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`

Consider a series defined as follows:

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

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

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:

$$1 + 2 + \ldots + (n-1) + n = \big(1 + 2 + \ldots + (n-1)\big) + n$$

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:

$$1^2 + 2^2 + \ldots + (n-1)^2 + n^2 = \big(1^2 + 2^2 + \ldots + (n-1)^2\big) + n^2$$

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 results of applying a given function f to the first *n* integers
- Give an alternative implementation of `sum_integers` and `sum_squares` using `sigma`

A tour of some higher-order functions
Generalizing the sum of the *n* first integers

Sum of *n* first integers 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 implemented as:

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

```
let rec siggma (n:int) (f:int → int):int=
 if n=0 then 0 (* implicitely we assume that f 0 = 0 *)
 else (f n) + siggma (n−1) f
```

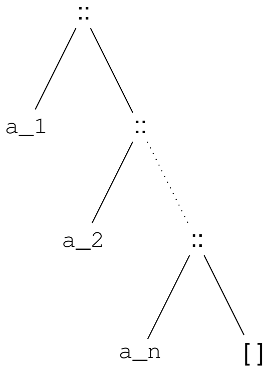
```
let sum_integers2 (n:int):int = siggma n (fun x → x)
```

```
let sum_squares2 (n:int):int = siggma n (fun x → x*x)
```

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`; ...; `a_n`]:



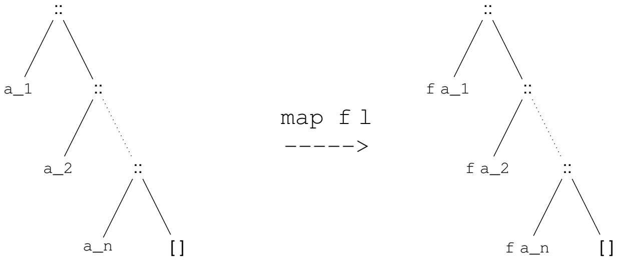
Graphic representation from Pierre Wiels and Xavier Leroy

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`

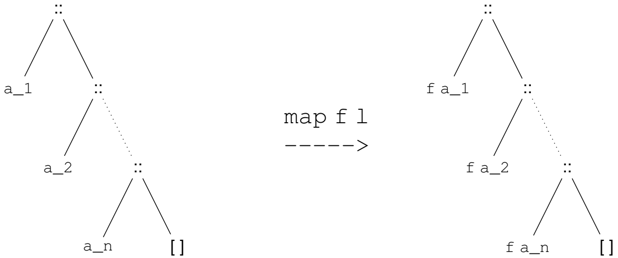
A tour of some higher-order functions

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) * Seq(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
```

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

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

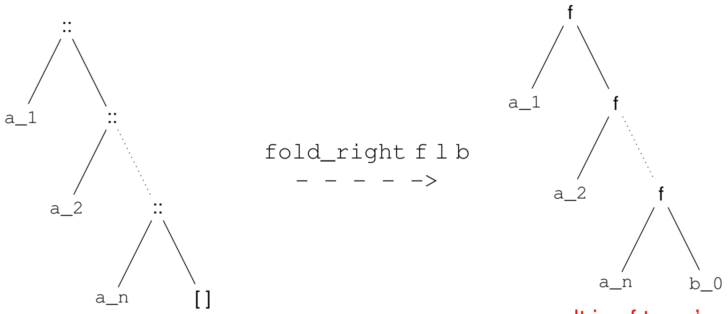
Lists: iterating a function on all elements on a list - function `fold_right`

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` 1)...))
- `concatenate` [`a1`;`a2`;...;`an`] = ^ `a1` (^ `a2` (... (^ `an` 0)...))

More generally, given:

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



$\rightarrow$ result is of type `'a`

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.

`fold_right f` [`a1`;...;`an`] `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

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`

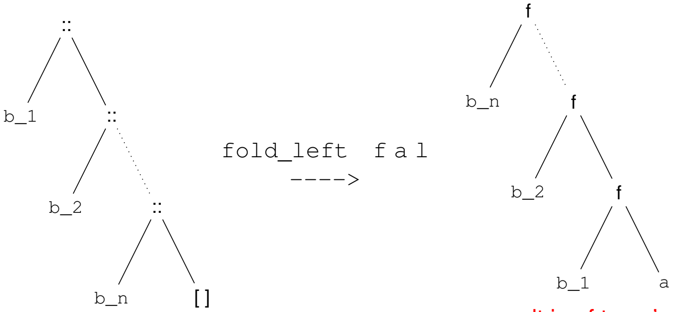
  Find 2 lists of inputs, so that the application of the function test_suite
  1. finds the bug
  2. does not find the bug

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



→ result is of type `'a`

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)
```

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`

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 `x1`,...,`xn` is actually a function that takes `x1` as a parameter and returns a function that takes `x2`,...,`xn` as parameters

The application

$$\texttt{f x1 x2 ... xn}$$

is actually a series of applications

$$\texttt{f (... (f x1) x2) ...) xn)}$$

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 `t1` $\to$ `t2` $\to$ ... $\to$ `tn` $\to$ `t`, and
- `xi` is of type `ti` for $i \in [1, j] \subseteq [1, n]$

Then `f x1 x2 ... xj` is of type `t(j+1)` $\to$ ... $\to$ `tn` $\to$ `t`

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 $c \in C$

Without currying:	With currying:
`f`: `tA` $*$ `tB` $\rightarrow$ `tC`	`f`: `tA` $\rightarrow$ `tB` $\rightarrow$ `tC`

`f` takes 1 argument: a pair
`f (a,b)` is of type `tC`
	`f` takes 2 arguments
	`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 □

Another way to learn the concepts of currying

www.ffconsultancy.com/ocaml/benefits/functional.html

A function of many arguments can be written as a function that accepts one argument and returns a function to consume the remaining arguments. This transformation is known as currying.

In OCaml, functions of many arguments are conventionally written in curried form. This is a stype of written functions.

Another way to learn the concepts of currying

www.ffconsultancy.com/ocaml/benefits/functional.html

Example: Raise a floating point number x to an integer power n could accept its arguments simultaneously as a 2-tuple (n, x):

```
# let rec pow(n, x) =
   if n=0 then 1. else x *. pow(n−1, x);;
val pow : int * float → float = <fun>
```

When written in curried form, this function accepts n and returns a function to raise the given x to the power n:

```
# let rec pow(n)(x) =
   if n=0 then 1. else x *. pow(n−1)(x);;
val pow : int → float → float = <fun>
```

For example, to create a function to cube a given number by applying only the first argument n=3:

```
# let cube = pow(3);;
val cube : float → float = <fun>
# cube 2.;;
− : float = 8.
```

Another way to learn the concepts of currying

Another example: define a plus function:

```
# let plus a b =
   a + b;;
val plus : int → int → int = <fun>
```

What is plus 2?

```
# plus 2;;
- : int → int = <fun>
```

This isn't an error. It's telling us that plus 2 is in fact a function, which takes an int and returns an int.
We experiment by first of all giving this mysterious function a name (f), and then trying it out on a few integers to see what it does:

```
# let f = plus 2;;
val f : int → int = <fun>
# f 10;;
- : int = 12
# f 15;;
- : int = 17
```

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