

# Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 0 - Sequence 5:

The *OCaml language*: a bird's eye view



# Taking the tour

Objective of this sequence

Present a few examples showcasing some of the features of the *OCaml* language.

- ▶ *safety* from **strong static typing** and **pattern matching**
- ▶ *conciseness* from **polymorphic typing** and **type inference**
- ▶ *expressiveness* from **higher order functions**

Disclaimer

It is a quick tour to give you *a taste* of the language.

- ▶ you are *not expected* to fully understand the examples right now...
- ▶ ... *you will understand everything*, and more, at the end of the course!

So hold tight, and let's go!

# Meeting the lists

In the following examples, we will use the `list` data structure.

In *OCaml*, lists are built-in

- ▶ `[]` is the *empty list*
- ▶ `a::l` is a *list* having `a` as first element, and the list `l` as rest

# Type inference

Let's write a function to sum all elements of an integer list :

```
# let rec suml =  
  function  
    []      -> 0  
  | a::rest -> a + (suml rest);;
```

We did not declare any type in our code...

```
val suml : int list -> int = <fun>
```

The *OCaml's* type checker *infers* the good type for us, *for free!*

# Strong static typing

All types are **computed** and **enforced** at compile time:

```
# sum1 [1;2;3];;
```

```
– int = 6
```

```
# sum1 ["1";"2";"3"];;
```

*Characters 6–9:*

```
sum1 ["1";"2";"3"];
```

*Error: This expression has type string but an expression was expected of type int*

*Well-typed programs cannot go wrong.*

*Robin Milner*

# Polymorphic types, and higher order

Let's generalise our function: `0` and `+` can be made into parameters:

```
# let rec suml =
  function
    []      -> 0
  | a::rest -> a + (suml rest);;

# let rec fold op e =
  function
    []      -> e
  | a::rest -> op a (fold op e rest);;
```

Again, we did not declare any type in our code...

```
val fold : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

The *OCaml's* type checker *infers a general type* for us, *for free!*

# Polymorphism and higher order at work

```
# fold ( + ) 0 [1;2;3;4;5];;
```

```
- int = 15
```

```
# fold ( * ) 1 [1;2;3;4;5];;
```

```
- int = 120
```

```
# fold ( ^ ) "" ["1";"2";"3"];;
```

```
- : string = "123"
```

```
# fold ( fun (x,y) a -> x + a ) 0 [(2,4);(3,5)];;
```

```
- : int = 5
```

# Pattern matching: ensuring all cases are handled

Let's write a function to remove all duplicates from a list of elements:

```
# let rec destutter =  
  function  
    | []                -> []  
    | x :: y :: rest ->  
      if x = y then destutter (y :: rest)  
      else x :: destutter (y :: rest) ;;
```

*Warning 8: this pattern-matching is not exhaustive.*

*Here is an example of a **value** that is not matched:*

```
_ :: []
```

```
val destutter : 'a list -> 'a list = <fun>
```

The compiler is telling us *which case we missed!*

Let's follow its advice...



# Pattern matching: ensuring all cases are handled

```
# let rec destutter =  
  function  
    | []                -> []  
    | x :: []          -> x :: []  
    | x :: y :: rest ->  
      if x = y then destutter (y :: rest)  
      else x :: destutter (y :: rest) ;;
```

```
val destutter : 'a list -> 'a list = <fun>
```

```
# destutter [1;1;2;2;2;3;1;4;2;2] ;;
```

```
- : int list = [1; 2; 3; 1; 4; 2]
```

# Conclusion

This was just a glimpse of the *OCaml* language and features.

Much more is in store for you in the rest of the course.