

# Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 2 - Sequence 3: Constructing and Observing Arrays



# Unbounded composite values

- ▶ A limitation of tuples and records: their sizes are statically bounded.
- ▶ Arrays allow to define composite values whose size is dynamically defined.
- ▶ For type-checking to remain simple, all array elements must have the same type.

# One array I

```
let p = [| 1; 2; 3 |];;  
# val p : int array = [|1; 2; 3|]
```

# A function producing arrays of multiple size I

```
let square x = x * x;;  
# val square : int -> int = <fun>  
let squares n = Array.init n square;;  
# val squares : int -> int array = <fun>  
let s1 = squares 5;;  
# val s1 : int array = [|0; 1; 4; 9; 16|]
```

# Syntax for array type

- ▶ The type of an array whose elements have `some_type` is  
`some_type array`
- ▶ `array` is a predefined type constructor.
- ▶ The standard library module `Array` provides functions over arrays.

# Syntax for array construction

- ▶ Arrays whose elements and sizes are known at compile-time are written:

```
[| some_expression; ...; some_expression |]
```

- ▶ The function `Array.make` expects an integer representing the size of the array and a value to initialize each component of the array.
- ▶ The function `Array.init` expects an integer representing the size of the array and a **function** to initialize each component of the array.
- ▶ The initialization function is given the index of the component and must return its value.
- ▶ `Array.length` returns the size of an array.

# Syntax to observe array cells

- ▶ To observe a specific component of an array using an **index**:

```
some_expression.(some_expression)
```

- ▶ Indexes of array `a` are taken between 0 and `Array.length a - 1`.
- ▶ To observe several components of an array, one can use **array patterns**:

```
[| some_pattern; ...; some_pattern |]
```

# Accessing array cells I

```
let swap a = [| a.(1); a.(0) |];;  
# val swap : 'a array -> 'a array = <fun>  
let b = swap [| 0; 1 |];;  
# val b : int array = [|1; 0|]  
let c = swap [| 0; 1; 2 |];;  
# val c : int array = [|1; 0|]
```



# Pattern matching over arrays I

```
let swap [| x; y |] = [| y; x |];;  
# Characters 9-32:  
  let swap [| x; y |] = [| y; x |];;  
    ~~~~~
```

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

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

```
[| |]  
val swap : 'a array -> 'a array = <fun>  
let t = swap [| 2; 1 |];;  
# val t : int array = [|1; 2|]  
let t = swap [| 2; 1; 0 |];;  
# Exception: Match_failure ("//toplevel//", 1, 9).
```

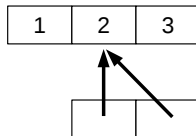
# In the machine

## Program

```
let p = [| 1; 2; 3 |]
```

```
let q = [| p; p |]
```

## Machine



- In memory, an array is a heap-allocated block.

# Pitfalls: Heterogeneous element types

- ▶ All the elements of an array must have the same type.

# Type clash between array components I

```
let a = [| true; 1 |];;
```

```
# Characters 17-18:
```

```
let a = [| true; 1 |];;  
           ^
```

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

# Pitfalls: Out of bound

- ▶ The compiler cannot ensure that all observation is valid.
- ▶ A negative index or an index greater than `Array.length a - 1` is an invalid observation of the array `a`.

# There is nothing outside I

```
let a = [| 0; 1; 2 |];;  
# val a : int array = [|0; 1; 2|]  
let x = a.(3);;  
# Exception: Invalid_argument "index_out_of_bounds".  
let y = a.(-1);;  
# Exception: Invalid_argument "index_out_of_bounds".
```