

Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 2 - Sequence 1: Constructing and Observing Tuples



Composite values

- ▶ Some values are naturally made of several components.
- ▶ Example:
 - ▶ A citizen identification = a name, a firstname, and a social security number.
 - ▶ A 2D coordinate = an abscissa, an ordinate.
- ▶ How can we **construct** and **observe** composite values?

2D coordinates I

```
let origin = (0, 0);;  
# val origin : int * int = (0, 0)  
let x_positive_limit = (max_int, 0);;  
# val x_positive_limit : int * int = (4611686018427387903, 0)  
let x_negative_limit = (min_int, 0);;  
# val x_negative_limit : int * int =  
    (-4611686018427387904, 0)
```

2D coordinates documented with types I

```
type point2D = int * int;;  
# type point2D = int * int  
let origin : point2D = (0, 0);;  
# val origin : point2D = (0, 0)  
let x_positive_limit : point2D = (max_int, 0);;  
# val x_positive_limit : point2D = (4611686018427387903, 0)  
let x_negative_limit : point2D = (min_int, 0);;  
# val x_negative_limit : point2D = (-4611686018427387904, 0)
```

Syntax for tuple construction

- ▶ The **type constructor** “*” constructs tuple types:

`some_type * ... * some_type`

- ▶ A tuple is constructed by separating its components with a comma “,”:

`(some_expression, ..., some_expression)`

- ▶ How to **observe** the components of a tuple?

Pattern matching

- ▶ **Patterns** describe how values are observed by the program.
- ▶ Patterns appear in `let`-bindings and as function arguments.
- ▶ We already saw the simplest form of pattern: identifiers.

```
let x = 6 * 3 in x
```

... can be read as “I observe the value of `6 * 3` by naming it `x`”.

- ▶ Another simple way to observe a value is to ignore it using a **wildcard** pattern:

```
let _ = 6 * 3 in 1
```

... can be read as “I ignore the value of `6 * 3`.”

Pattern matching tuples

- ▶ Patterns can be composed to describe the observation of tuples:

```
let (x, _) = (6 * 3, 2) in x
```

... can be read as:

- ▶ “I observe the first component of $(6 * 3, 2)$ by naming it x ”
- ▶ and “I ignore the second component of $(6 * 3, 2)$ ”.

Extract the two components of a pair I

```
let a = (3 * 6, 4 * 6);;  
# val a : int * int = (18, 24)  
let (x, _) = a;;  
# val x : int = 18  
let abscissa (x, _) = x;;  
# val abscissa : 'a * 'b -> 'a = <fun>  
let ordinate (_, y) = y;;  
# val ordinate : 'a * 'b -> 'b = <fun>
```


Syntax for tuple patterns

- ▶ A pattern that matches a tuple has the form:

`(some_pattern, ..., some_pattern)`

- ▶ The number of subpatterns must be equal to the number of tuple components.
- ▶ An identifier can only occur once in a pattern.

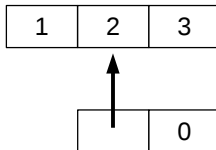
In the machine

Program

let p = (1, 2, 3)

let q = (p, 0)

Machine



Program

let p = (1, 2, 3)

let q = (p, p)

- ▶ A tuple is represented by a **heap-allocated block**.
- ▶ The program holds a pointer to this block.
- ▶ This pointer can be **shared**.

Structural equality VS physical equality

- ▶ In *OCaml*, the operator `=` implements **structural equality**.
- ▶ Two values are structurally equal if they have the same content.
- ▶ The operator `==` implements **physical equality**.
- ▶ Two values are physically equal if they are stored in the same memory location.

Structural equality VS physical equality I

```
let x = (1, 2);;  
# val x : int * int = (1, 2)  
let y = (1, 2);;  
# val y : int * int = (1, 2)  
let z = x;;  
# val z : int * int = (1, 2)  
let x_is_structural_equal_to_y = (x = y);;  
# val x_is_structural_equal_to_y : bool = true  
let x_is_not_physically_equal_to_y = (x == y);;  
# val x_is_not_physically_equal_to_y : bool = false  
let x_is_physically_equal_to_z = (x == z);;  
# val x_is_physically_equal_to_z : bool = true
```

Pitfalls: Ill-formed patterns

- ▶ Invalid arity.
- ▶ Nonlinear patterns.
- ▶ These errors are caught by the compiler!

Ill-formed patterns I

```
let (x, _) = (1, 2, 3);;
```

```
# Characters 13-22:
```

```
  let (x, _) = (1, 2, 3);;  
          ^^^^^^^^^
```

Error: This expression has **type** 'a * 'b * 'c
but an expression was expected **of type** 'd * 'e

```
let (x, x, y) = (1, 2, 3);;
```

```
# Characters 8-9:
```

```
  let (x, x, y) = (1, 2, 3);;  
      ^
```

Error: Variable x is bound several times **in** this matching

Pitfalls: Semantically invalid projection

- ▶ Definition-by-position is error-prone.

A semantically invalid projection I

```
let abscissa (x, y) = y;;  
# val abscissa : 'a * 'b -> 'b = <fun>  
let ordinate (x, y) = x;;  
# val ordinate : 'a * 'b -> 'a = <fun>
```


Pitfalls: Semantically invalid projection

What's next?

Records will help us avoid such errors.