

INF231:  
Functional Algorithmic and Programming  
Lecture 3: Advanced types

Academic Year 2020 - 2021

$f(x)$



# Modelling information/concepts

What is modelling?

Why modelling?

How to model?

- ▶ defining specific data types
- ▶ defining functions manipulating these data types

## Defining a type

The general form

```
type t = ... (* possibly with constraints *)
```

Now we are going to see how we can define some more complex types using existing types...

# Outline

Synonym types

Enumerated types

Product types

Union/Sum types

Case study: Modelling 4 card games

## Defining a synonym type

Motivations:

- ▶ context-specific types
- ▶ easier to remember
- ▶ re-use

General syntax:

```
type new_type = existing_type  
    (* possibly with informative usage constraints *)
```

### Example (Soldes)

- ▶ `type price = float (* > 0 *)`
- ▶ `type rate = int (* 0, ..., 99 *)`
- ▶ Defining a function to reduce prices:
  - ▶ Description: `reducedPrice(p,r)` is the price `p` reduced by `r%`
  - ▶ Profile: `reducedPrice: price * rate → price`
  - ▶ Examples: `reducedPrice(100., 25) = 75.`

(note that it is more meaningful than the “anonymous signature”  
`reducedPrice: float * int → float`)

# Outline

Synonym types

**Enumerated types**

Product types

Union/Sum types

Case study: Modelling 4 card games

## Enumerated types

Motivation: How can we model/define/use:

- ▶ the family of a card? {♠, ♥, ♦, ♣}
- ▶ the color of a card? {black, white}

From a mathematical point of view: sets defined extensively  
↔ i.e., by an *explicit enumeration*

Defining an **enumerated type** in OCaml:

```
type new_type = Value_1 | Value_2 | ... | Value_n
```

### Remark

- ▶ Capital letters are mandatory
- ▶ `new_type` is said to be an *enumerated type*
- ▶ `Value_1, ..., Value_n` are said to be *symbolic constants*
- ▶ `Value_1, ..., Value_n` are of type `new_type`
- ▶ Implicit order between constants (consequence of the definition)



# Enumerated types: Some examples

Painting / Modelling a card game

## Example (Some paint colors)

```
type paint =  
  | Red  
  | Blue  
  | Yellow
```

## Example (Types of a Card game)

```
type family = Spade | Heart  
             | Diamond | Club  
type color = White | Black
```

DEMO: types of card game

## Example (Color of a family)

Returning the color associated to a family card

- ▶ **Description:** `colorFamily` returns the family of a given card.
  - ▶ Heart and Diamond are associated to White
  - ▶ Spade and Club are associated to Black
- ▶ **Signature:** `colorFamily: family → color`
- ▶ **Examples:** `colorFamily Spade = Black, ...`

DEMO: Implementation of `colorFamily`



## Back to the language constructs: **pattern-matching**

Your best friend

One of the most powerful feature of OCaml (and functional languages)

Pattern-matching: computation by **case analysis**

Specified by the following syntax:

```
match expression with
| pattern_1 → expression_1
| pattern_2 → expression_2
  ...
| pattern_n → expression_n
```

Meaning:

- ▶ `expression` is matched against the patterns, i.e., its value is evaluated and then compared to the patterns **in order**  
↪ “matching” depends on the type of `expression`!
- ▶ the expression associated to the first matching pattern is returned

**Remark**

- ▶ First vertical bar is optional
- ▶ may use `_` as a *wild-card* (should be the last pattern)



## (Pattern) Matching on an example

The card game

### Example (colorFamily using if...then...else)

```
let colorFamily (f:family):color =  
  if (f=Spade || f = Club) then Black  
  else (* necessarily f = Heart || f = Diamond *)  
    White
```

### Example (colorFamily using pattern-matching)

```
let colorFamily (f:family):color =  
  match f with  
  | Spade → Black  
  | Club → Black  
  | Heart → White  
  | Diamond → White
```

## (Pattern) Matching on an example

The card game with more concise pattern-matching

Example (colorFamily using a more concise pattern-matching)

```
let colorFamily (f:family):color =  
  match f with  
    Spade | Club → Black  
    | Heart | Diamond → White
```

Example (colorFamily using an even more concise pattern-matching)

```
let colorFamily (f:family):color =  
  match f with  
    Spade | Club → Black  
    | _ → White
```

Example (colorFamily using an even even more concise pattern-matching)

```
let colorFamily = function | Spade | Club → Black  
  | _ → White
```

## Pattern-matching for enumerated types

To the enumerated type

```
type newtype = Value_1 | Value_2 | ... | Value_n
```

is associated the pattern matching

```
match expression with (* expression is of type newtype *)  
  | Value_1 → expression_1  
  | Value_2 → expression_2  
  ...  
  | Value_n → expression_n
```

### Rules

- ▶ Pattern-matching “follows” the definition of the type (not necessarily with the same order)
- ▶  $expression_i$  for  $i \in \{1, \dots, n\}$  should be of the same type
- ▶ Should be **exhaustive** (or use the wild-card symbol `_`)

```
match expression with  
  | Value_1 → expression_1  
  ...  
  | _ → expression
```

## Let's practice enumerated types

### Exercise

- ▶ Define the enumerated type `month` which represents the twelve months of the year
- ▶ Define the function `nb_of_days: month → int` which associates to each month its number of days

## Matching (also) works (more or less) with (some) predefined types

Pattern-matching is a generalization of the `if...then...else...`

↪ works with existing/predefined types: `int`, `bool`, `float`, `char`, `string`

### Example (Is an integer an even number?)

```
let is_even (n : int) : bool =  
  match n with  
  | 0 → true  
  | 1 → false  
  | 2 → true  
  | n → if n mod 2 = 0 then true else false
```

### Example (Is a character in upper case?)

```
let is_uppercase (c : char) = match c with  
  'A' → true  
  | 'B' → true  
  | ... (* 23 conditions *)  
  | 'Z' → true  
  | c → false
```

### Example (Matching with floats is dangerous)

```
match 4.3 -. 1.2 with  
  3.1 → true  
  _   → false           ~ returns false
```

## Some shortcuts with pattern-matching

For enumerated types

“Disjuncting equivalent patterns”:

```
match something with
```

```
...  
| p1 → v  
| p2 → v  
...  
| pm → v  
....
```

can be shortened into

```
match something with
```

```
...  
| p1 | p2 | pm → v  
....
```

Example (“Disjuncting equivalent patterns”)

```
let is_uppercase (c:char) = match c with  
  'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M'  
  | 'N' | 'O' | 'P' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' → true  
  | c → false
```

## Some shortcuts with pattern-matching - ctd

For characters

“Leveraging the order between characters”:

```
match something with
  ...
  | p1 .. pm → v
  ...
  | p1 → v
  | p2 → v
  ...
  | pm → v
  ....

~>

match something with
  ...
  | pm .. p1 → v
  ....
```

where  $p_1, \dots, p_m$  are *consecutive* characters and  $p_1$  and  $p_m$  are the minimal and the maximal characters (not necessarily in this order)

Example (“Leveraging the order between the elements of characters”)

```
let is_uppercase (c:char)
  = match c with
    'A' .. 'Z' → true
    | c → false

or

let is_uppercase (c:char)
  = match c with
    'Z' .. 'A' → true
    | c → false
```



# Outline

Synonym types

Enumerated types

**Product types**

Union/Sum types

Case study: Modelling 4 card games

## Product type: motivating example(s) and connection with maths

### Example (Some complex numbers)

How can we **model** complex numbers?

In maths, we define:

$$\mathbb{C} = \{a + ib \mid a \in \mathbb{R}, b \in \mathbb{R}\}$$

<b>z</b>	<b>a</b>	<b>b</b>
$3.0 + i * 2.5$	3.0	2.5
$12.0 + i * 1.5$	12.0	1.5
$(1.0 + i) * (1.0 - i)$		

Actually, we could also define:

$$\mathbb{C} = \mathbb{R} \times \mathbb{R}$$

The operation  $\times$  is the **Cartesian product of sets**

### Example (Defining card)

Same reasoning can be followed if we want to define the type of a card. . .

## (Cartesian) Product (of) type

We can build **Cartesian product** of types, i.e., pairs of object of different types:

Type Constructor	Value Constructors
$\alpha * \beta$	$\bullet, \bullet$
<code>int*int</code>	1,2
<code>int*float</code>	1,2.0

DEMO: A couple of pairs

Defining new product types:

```
type new_type = existing_type1 * existing_type2
```

Two basic operations on pairs:

▶ `fst (•1, •2) = •1`

▶ `snd (•1, •2) = •2`

Deconstruction on pairs (hidden pattern matching):

```
let (x1,x2) = (v1,v2) in expression_using_x1_and_x2
```

↪ defines the identifiers `x1` and `x2` locally

DEMO: Product types

# General Cartesian product of types

Same principle

Can be generalized to  $n$ -tuples:

- ▶ type definition/construction:

```
let my_type = type1 * type2 * ... * typen
```

- ▶ value *construction*:  $v_1, v_2, \dots, v_n$

- ▶ value *deconstruction*:

```
let (x1, ..., xn) = (v1, ..., vn) in expression  
(* expression is depending on x1, ..., xn *)
```

DEMO: Generalized Product types

## Let's practice product type

### Exercise: Getting familiar with tuples

- ▶ Define the type `pair_of_int` which implements pairs of integers
- ▶ Define the function `swap` which swaps the integers in a `pair_of_int`
- ▶ Implement a function `my_fst` which behaves as the predefined function `fst` on `pairs_of_int`

### Exercise on Complex numbers

- ▶ Define the type `complex` which corresponds to complex numbers
- ▶ Define function `real_part` of type `complex → float` which returns the real part of a complex number
- ▶ Define function `im_part` of type `complex → float` which returns the imaginary part of a complex number
- ▶ Define function `conjugation: complex → complex`  
Remainder: the conjugation of  $a + b.i$  is  $a - b.i$

# Let's practice more

## Geometry and vectors

### Exercise on vectors

- ▶ Define the type `vect` which corresponds to vectors in the plane
- ▶ Define the function `sum : vect → vect → vect` which performs the sum of two vectors
- ▶ What is the type of the function which implements the scalar product?
- ▶ Implement a function which performs the scalar product of two vectors  
Remainder: scalar product of two vectors  $\vec{u}, \vec{v} : \|\vec{u}\| \cdot \|\vec{v}\| \cdot \cos(\vec{u}, \vec{v})$   
with  $\cos(\vec{u}, \vec{v}) = \frac{u_x \cdot v_x + u_y \cdot v_y}{\|\vec{u}\| \cdot \|\vec{v}\|}$
- ▶ A vector can represent the position of a point in the plane. The rotation of angle  $\theta$  of a point of coordinates  $(x, y)$  around the origin is expressed by the formula:

$$\begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Implement the function `rotation : float → vect → vect` such that `rotation angle v` makes the vector designated by `v` rotating of an angle `angle`

# Outline

Synonym types

Enumerated types

Product types

**Union/Sum types**

Case study: Modelling 4 card games

# Motivating union types

Mixing carrots and cabbage

... in the context of OCaml type system

Some concepts that we cannot model yet:

- ▶ How to build a type `figure` which can represent circles, triangles, quadrilaterals?
- ▶ How to build a type which allows to represent a full color palette ?



- ▶ How to build a card game which can represent various games?



# Back to the paint

Introducing Union types through an example

Type definition	Filtering
<pre>type paint =   Blue     Yellow     Red</pre>	<pre>let is_blue (p : paint) : bool =   match p with     Blue → true     Yellow → false     Red → false</pre>

**Remark** The type `paint` contains *three constant constructors*



*How can we add to the set of paints, some new paints that do not have a name, but only reference number?*

# Back to the paint

Introducing Union types through an example

Type definition	Filtering
<pre>type paint =     Blue     Yellow     Red     Number of int</pre>	<pre>let is_blue (p : paint) : bool =   match p with     Blue → true     Yellow → false     Red → false     Number i → false</pre>

## Remark

- ▶ Type `paint` has 3 constant constructors and one **non constant constructor**.
- ▶ Number 14 represents the paint numbered 14 (in an imaginary catalogue)



# Back to the paint

Introducing Union types through an example

Type definition	Filtering
<pre>type paint =     Blue     Yellow     Red (* palette RGB *)     RGB of int * int * int</pre>	<pre>let is_blue (p : paint) : bool =   match p with     Blue → true     Yellow → false     Red → false     RGB (r,g,b) → r = 0 &amp;&amp; g = 0 &amp;&amp; b = 255</pre>

- ▶ Type `paint` has three constant constructors and two non-constant constructors
- ▶ `RGB(255,0,0)` corresponds to red
- ▶ `RGB(255,255,0)` corresponds to yellow
- ▶ ...

# Union types (aka union type, tagged union, algebraic data types)

The general form

Syntax of **union types**:

```
type new_type =  
  | Identifier_1 of type_1  
  | Identifier_2 of type_2  
  ...  
  | Identifier_n of type_n
```

Note that:

- ▶  $\text{Identifier}_i, i \in [1, n]$ , is an explicit name called a **constructor**
- ▶ the definition “of type<sub>*i*</sub>” is optional
- ▶  $\text{type}_i, i \in [1, n]$ , can be any (existing) type
- ▶ Constructor name must be capitalized

Expression Declaration (of some type  $t$ ):

```
let expression = Identifier v
```

(if  $\text{Identifier}$  of  $tt$  is a constructor of type  $t$  and  $v$  is a value of type  $tt$ )

**Remark**

- ▶ Union types are a generalization of enumerated types



## An example: Generalization of `int` and `float`

Having two different sets of operations for `int` and `float` is sometimes annoying

Let's define  $\text{Numbers} = \mathbb{R} \cup \mathbb{N}$

```
type numbers = INTEGER of int | REAL of float
```

(INTEGER, REAL sont des constructeurs de type)

Let's define additions on two numbers:

```
let add ((nb1,nb2):number*number) : number = match (nb1,nb2) with
| (INTEGER(n1), INTEGER(n2)) → INTEGER(n1 + n2)
| (INTEGER(n), REAL(r)) → REAL( (float_of_int n) +. r)
| (REAL(r), INTEGER(n)) → REAL( (float_of_int n) +. r)
| (REAL(r1), REAL(r2)) → REAL(r1 +. r2)
```

**Remark** Has some advantages and disadvantages



## Another example: Geometry

Type definition	Filtering
<pre>type pt = float * float  type figure =     Rectangle of pt * pt     Circle of pt * float     Triangle of pt * pt * pt</pre>	<pre>let perimeter (f : figure) : float =   match f with     Rectangle (p1, 2) → ...     Circle (_, r) → ...     Triangle (p1, p2, p3) → ...</pre>
<pre>let p1 = 1.0, 2.0 and p2 = 3.9, 2.7 in Rectangle (p1,p2) let p1 = (1.3, 2.9) in Circle (p1,3.6)</pre>	

### Exercise

- ▶ Define the function `distance: pt → pt → float`
- ▶ The area of any triangle of edge lengths  $a$ ,  $b$ ,  $c$  is computed using the Héron's formula:

$$A = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)} \quad \text{with} \quad s = \frac{1}{2} \cdot (a + b + c)$$

Define the function `area: figure → float`

## Remark: *distinguish* constructors and functions

Constructors and functions take a value of some type and return another value of some other type.

A *function*:

- ▶ performs a computation
- ▶ cannot be used in pattern matching: the value of all functions is `<fun>`

A *type constructor*:

- ▶ constructs a value
- ▶ can be used in a pattern-matching

## Remark: Difference between union and sum

There is actually a slight difference between union and sum

Consider two sets  $E$  and  $F$ :

Union	Sum
$E \cup F$	$\{\text{FromE}(x) \mid x \in E\} \cup \{\text{FromF}(x) \mid x \in F\}$
“everything is merged/mixed”	“elements are decorated” and then merged

Second solution is less ambiguous and then preferred by computers



# Card Game

Your choice



Playing cards:



Images from Wikipedia, Licence CC

# Conclusion

## Summary:

- ▶ Richer types:

Type	Why?
synonym types	informative type names
enumerated types	Finite set of constants
product types	Cartesian product
sum types	Set Union

- ▶ Using filtering and pattern matching to define more complex functions (for each of these types)

## Exercise

Find a (personal) example of objects that can be naturally modelled as a union type. Propose/Invent a function using this type.