Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 3 - Sequence 5: Advanced topics about data types

Precise typing

- **A sum type with only one constructor** can be useful to discriminate between two types that are structurally equivalent but semantically different

Euros are not dollars I

```
type euro = Euro of float;;
# type euro = Euro of float
type dollar = Dollar of float;;
# type dollar = Dollar of float
let euro_of_dollar (Dollar d) = Euro (d /. 1.33);;
# val euro_of_dollar : dollar -> euro = <fun>
let x = Dollar 4.;;
# val x : dollar = Dollar 4.
let y = Euro 5.;;
# val y : euro = Euro 5.
```

Euros are not dollars II

```
let invalid_comparison = (x < y);;
# Characters 30-31:
  let invalid_comparison = (x < y);;
                                ^
Error: This expression has type euro but an expression was expected
    of type
         dollar
let valid_comparison = (euro_of_dollar x < y);;
# val valid_comparison : bool = true
```

Disjunctive patterns

- Sometimes, the same code is duplicated in several branches.
- **or**-patterns allow you to **factorize these branches into a unique branch**.
- "`some_pattern_1 | some_pattern_2`" corresponds to the observation of `some_pattern_1` **or** `some_pattern_2`.
- `some_pattern_1` and `some_pattern_2` must contain the same identifiers.

Disjunctive pattern I

```
let remove_zero_or_one_head = function
  | 0 :: xs -> xs
  | 1 :: xs -> xs
  | l -> l;;
# val remove_zero_or_one_head : int list -> int list = <fun>
let remove_zero_or_one_head' = function
  | 0 :: xs | 1 :: xs -> xs
  | l -> l;;
# val remove_zero_or_one_head' : int list -> int list = <fun>
let remove_zero_or_one_head'' = function
  | (0 | 1) :: xs -> xs
  | l -> l;;
# val remove_zero_or_one_head'' : int list -> int list =
  <fun>
```

as-patterns

- It is sometimes convenient to **name a matched component**.
- The pattern "`some_pattern` **`as`** `x`" is read as
  "If the value can be observed using `some_pattern`, name it `x`."

as-pattern I

```
let rec duplicate_head_at_the_end = function
  | [] -> []
  | (x :: _) as l -> l @ [x];;
# val duplicate_head_at_the_end : 'a list -> 'a list = <fun>
let l = duplicate_head_at_the_end [1;2;3];;
# val l : int list = [1; 2; 3; 1]
```

Constrained pattern matching branch using `when`

- A boolean expression, called a **guard**, can add an extra constraint to a pattern.
- This guard is introduced by the keyword `when`.

Guarded patterns I

```
let rec push_max_at_the_end = function
  | ([] | [_]) as l -> l
  | x :: ((y :: _) as l) when x <= y -> x :: push_max_at_the_end l
  | x :: y :: ys  -> y :: push_max_at_the_end (x :: ys);;
# val push_max_at_the_end : 'a list -> 'a list = <fun>
```

Other kinds of types

- There are advanced features of the type system that we did not show:
  - Objects
  - First-class modules
  - Polymorphic variants
  - Generalized algebraic datatypes

Next week, you will learn how to write
higher-order programs
over all the types we have seen so far!