Introduction to Functional Programming in *OCaml*

Roberto Di Cosmo, Yann Régis-Gianas, Ralf Treinen

Week 3 - Sequence 4: Polymorphic algebraic datatypes

Type parametric programming

- `list` is **a type constructor parameterized by the type of the elements**.
- Most of the functions over lists do not depend on the type of the elements.
- Hence, the module `List` contains only **polymorphic functions**.
- These are written once and for all, maximizing **code reuse**.

The programmer can define
her **own parameterized types**.

A type constructor for optional values I

```
(* The following type is predefined. *)
type 'a option =
  | None
  | Some of 'a;;
# type 'a option = None | Some of 'a

let o1 = Some 42;;
# val o1 : int option = Some 42
let o2 = None;;
# val o2 : 'a option = None
```

A type constructor for the disjoint union of two types I

```
type ('a, 'b) either =
  | Left  of 'a
  | Right of 'b;;
# type ('a, 'b) either = Left of 'a | Right of 'b
type square = { dimension : int };;
# type square = { dimension : int; }
type circle = { radius : int };;
# type circle = { radius : int; }
type shape = (square, circle) either;;
# type shape = (square, circle) either
```

A generic binary search tree I

```
type 'a bst =
  | Empty
  | Node of 'a bst * 'a * 'a bst;;
# type 'a bst = Empty | Node of 'a bst * 'a * 'a bst
```

A generic binary search tree II

```
let rec find_max = function
  | Empty -> assert false
  | Node (_, x, Empty) -> x
  | Node (_, x, r) -> find_max r;;
# val find_max : 'a bst -> 'a = <fun>
```

A generic binary search tree III

```
let rec insert x = function
  | Empty -> Node (Empty, x, Empty)
  | Node (l, y, r) ->
    if x = y then Node (l, y, r)
    else if x < y then Node (insert x l, y, r)
    else Node (l, y, insert x r);;
# val insert : 'a -> 'a bst -> 'a bst = <fun>
```

A generic binary search tree IV

```
let rec delete x = function
  | Empty ->
    Empty
  | Node (l, y, r) ->
    if x = y then join l r
    else if x < y then Node (delete x l, y, r)
    else Node (l, y, delete x r)
and join l r =
  match l, r with
    | Empty, r -> r
    | l, Empty -> l
    | l, r -> let m = find_max l in Node (delete m l, m, r);;
# val delete : 'a -> 'a bst -> 'a bst = <fun>
val join : 'a bst -> 'a bst -> 'a bst = <fun>
```

A generic binary search tree V

```
type contact = { name : string; phone_number : int }
type database = contact bst;;
# type contact = { name : string; phone_number : int; }
type database = contact bst
```

Syntax for parameterized types

- To declare a parameterized type:
  type `('a1, ..., 'aN) some_type_identifier = some_type`

- The **type variables** `'a1, ..., 'aN` represent unknown types.
- They can appear in `some_type`.
- To instantiate a parameterized type, we write:
  `(some_type, ..., some_type) some_type_identifier`

Pitfalls

- The arity of the type constructor must be respected.
- The type variables must be declared.

Invalid type application I

```
type ('a, 'b) t = 'a * 'b * 'a;;
# type ('a, 'b) t = 'a * 'b * 'a
type u = (int, bool, string) t;;
# Characters 9-30:
  type u = (int, bool, string) t;;
           ^^^^^^^^^^^^^^^^^^^^^

Error: The type constructor t expects 2 argument(s),
       but is here applied to 3 argument(s)
```

Unbound type variable I

```
type 'a t = 'a * 'b;;
# Characters 17-19:
  type 'a t = 'a * 'b;;
                   ^^
Error: Unbound type parameter 'b
```