OCaml Types

Jibesh Patra, Soumyajit Dey

Week 3 – Software Engineering – CS20202 – Spring 2026 – IIT Kharagpur

More Types

A Program to find the Last Element of a List

```
let rec last lst =
 match lst with
 | [] -> None
 | [ x ] -> Some x
 | _ :: t -> last t;;

last [1; 2; 3; 5];;
last [];;
```

```
let rec last = function
 | [] -> None
 | [ x ] -> Some x
 | _ :: t -> last t;;

last [1; 2; 3; 5];;
last [];;
```

```
val last : 'a list -> 'a option = <fun>
- : int option = Some 5
- : 'a option = None
```

- An option is used to express that a value might or might not be present.
- Two value constructors:
  - None
  - Some

Unwrapping Option Value

```
let rec last = function
| [] -> None
| [ x ] -> Some x
| _ :: t -> last t;;

let last_val_of_list input_lst = match
last input_lst with
| Some v -> v
| None -> -1;;

last_val_of_list [1; 2; 3; 5];;
last_val_of_list [];;
```

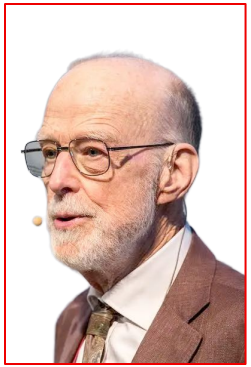
Using the value constructors to destructure.

```
val last : 'a list -> 'a option = <fun>
val last_val_of_list : int list -> int = <fun>
- : int = 5
- : int = -1
```

Creator of `null`

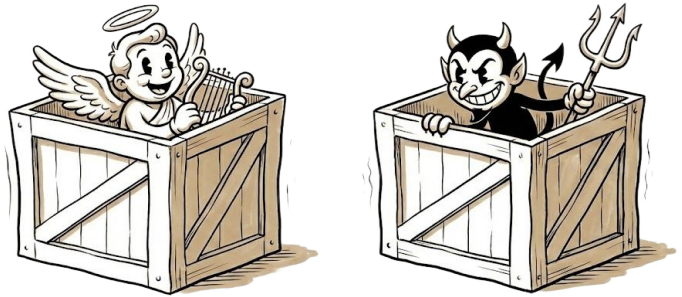
- Tony Hoare is the creator of null reference.

> *"Null references were created in 1964 - how much have they cost? Less or more than a billion dollars?"*



Result Type

- Options are not always sufficient.
- Used to express a function’s outcome.
- The outcome can be either
  - success
  - failure
- Exactly two ways to build a result value: `Ok` or `Error`



Result Type

```
let rec last = function
  | [] -> Error "Error! no last element found"
  | [x] -> Ok x
  | _ :: t -> last t;;
let last_val_of_list input_lst =
  match last input_lst with
  | Ok v -> v
  | Error msg ->
      print_endline msg;
      -1;;

last_val_of_list [1; 2; 3; 5];;
last_val_of_list [];;
```

```
val last : 'a list -> ('a, string) result = <fun>
val last_val_of_list : int list -> int = <fun>
- : int = 5
Error! no last element found
- : int = -1
```

Unit Type

- In OCaml, the phrases of the language are used for expressing values – *expression*.
- In some languages, the phrases of the language can be used to:
  - express values – expression.
  - command – statement. → Imperative programming
- We may have some expressions that need not compute any meaningful value.
- OCaml requires every expression to have a value.
- A special value – `()`.
- The `unit` type has only one value `()` and nothing else.

Unit Type Example (1/3)

```
let say_hello  = print_endline "Hello, World!";;
```

```
Hello, World!
val say_hello : unit = ()
```

```
let say_hello () = print_endline "Hello, World!";;

say_hello;;

say_hello ();;
```

```
val say_hello : unit -> unit = <fun>

- : unit -> unit = <fun>

Hello, World!
- : unit = ()
```

Optional Arguments Revisited

```
let bump ?(step = 1) x = x + step;;

bump 20;;
bump ~step:2 20;;
```

```
val bump : ?step:int -> int -> int = <fun>

- : int = 21
- : int = 22
```

- A function taking some optional arguments must also take at least one non-optional argument.

Unit Type Example (2/3)

```
let say_hello  ~(name:string)  ~(salutation:string) ?(course:string="SE") = "Hello, " ^
salutation ^" "^ name ^ ". Welcome to " ^ course;;

say_hello ~name: "Spiderman" ~salutation: "Mr";;
```

```
1 | let say_hello  ~(name:string)  ~(salutation:string) ?(course:string="SE") = "Hello, "
^ salutation ^" "^ name ^ ". Welcome to " ^ course;;
                                                              ^^^^^^
Warning 16 [unerasable-optional-argument]: this optional argument cannot be erased.

val say_hello : name:string -> salutation:string -> ?course:string -> string =
  <fun>

- : ?course:string -> string = <fun>
```

Unit Type Example (3/3)

```
let say_hello  ~(name:string)  ~(salutation:string) ?(course:string="SE") () = "Hello, "
^ salutation ^" "^ name ^ ". Welcome to " ^ course;;

say_hello ~name: "Spiderman" ~salutation: "Mr" ();;
```

```
val say_hello :
  name:string -> salutation:string -> ?course:string -> unit -> string = <fun>


- : string = "Hello, Mr Spiderman. Welcome to SE"
```

User-defined Types

Three Cases of User-Defined Types

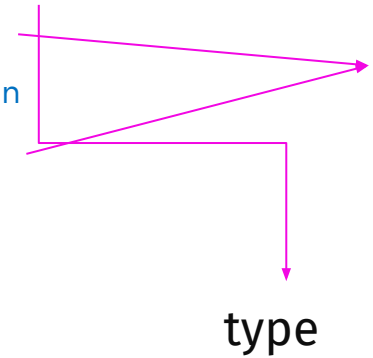
- Variants
- Records
- Aliases

General form of all user-defined types is: `type … = …`

Variant

A data type representing a value that is one of several possibilities.

```
type color =
  | Red
  | Green
  | Blue
```



value constructor (must be uppercase)

type

```
type color = Red | Green | Blue
```

Pattern Matching – Variants

```
type color =
| Red
| Green
| Blue;;

let hex = function
| Red -> "#FF0000"
| Green -> "#00FF00"
| Blue -> "#0000FF";;

hex Green;;
```

Using the value constructors to destructure.

```
type color = Red | Green | Blue

val hex : color -> string = <fun>

- : string = "#00FF00"
```

Value Constructor with Data

```
type shape =
  | Circle of float
  | Rectangle of float * float
  | Square of float
  | Point
  | Triangle of float * float
let area = function
  | Point -> 0.0
  | Circle r -> 3.1415 *. r *. r
  | Rectangle (w, h) -> w *. h
  | Square side -> side *. side
  | Triangle (base, height) -> 0.5 *. base *. height
```

```
let s1 = Circle 2.5
let s2 = Rectangle (3.0, 4.0)
let s3 = Point
let a1 = area s1
let a3 = area s3
```

```
..
val s1 : shape = Circle 2.5
val s2 : shape = Rectangle (3., 4.)
val s3 : shape = Point

val a1 : float = 19.6343750000000021
val a3 : float = 0.
```

Recursive Variants

```
type shape =
  | Circle of float
  | Rectangle of float * float
  | Point
  | Shapes of shape list
```

```
let s1 = Circle 2.4
let s2 = Rectangle (1.2, 2.0)
```

```
let shapes = Shapes [s1; Rectangle (2.4, 3.4); s2]
```

```
..
val s1 : shape = Circle 2.4
val s2 : shape = Rectangle (1.2, 2.)
val shapes : shape =
  Shapes [Circle 2.4; Rectangle (2.4, 3.4); Rectangle
(1.2, 2.)]
```

A variant definition referring to itself is recursive.

This is the end