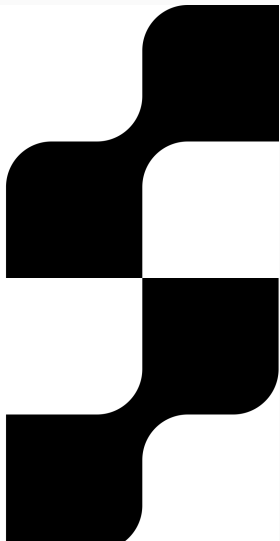


Introduction to OCaml

Richard Bonichon

20200625



Background

The Categorical Abstract Machine Language

OCaml is part of the ML family, like SML (big brother) or F# (little brother).

1973 (classic) ML [7]

1987-1992 Heavy CAML (LISP-based implementation)

1990-1991 Caml Light

1996 Objective Caml 1.00

2011 Objective Caml becomes OCaml

2020 OCaml 4.10.0 (Feb 21)

OCaml : an open-minded functional language

OCaml is **not** a pure functional language

- imperative programming is very much part of the nominal toolbox
- OOP too when it is the right fit.

Bytecode and native code support

2 compilers for the price of one:

ocamlc a bytecode compiler to a stack-based

- its interpreter `ocamlrun` works anywhere you have a C compiler

ocamlopt a native code compiler

- supports x86 (32/64), ARM (v5-v8), PowerPC (32/64) and ... SPARC !
- RISC-V will be in 4.11

- Academic circles
 - academia** INRIA, Berkeley, CEA, CMU, UArizona, UPenn
 - SMEs** OCamlPro/Origin Labs, Nomadic Labs, TrustInSoft, Tarides,
- Financial “institutions” – Bloomberg, Jane Street, Lexifi, SimCorp
- Facebook (ReasonML, Infer)
- Atos, AbsInt
- Indirect users: Airbus (Astrée, Frama-C, Fluctuat), EDF, ...
- ...

Natural application fields

- compilers
- program analysis
- theorem proving
- symbolic computations

Tooling (as of 2020)

compiler last release is 4.10 (2020-02-21)

merlin context-sensitive completion for OCaml (in Vim, Emacs, VSCode, ...)

A **very** nice tool which has changed the life of most OCaml developers, and it's editor-agnostic !

dune newest contender in dedicated build systems
Subsumes OCamlMakefile, omake, ocamlbuild.

OPAM 1.0 in 2013, current is 2.0.7 (2020-04-21)

A source-based package manager for OCaml software.

Emacs or another editor

<https://opam.ocaml.org/blog/turn-your-editor-into-an-ocaml-ide/>

Building blocks

let-bindings

```
1 | let add x y = x + y (* or let add = ( + ) *)
```

```
1 | val add : int -> int -> int = <fun>
```

```
1 | let simple_main () =  
2 |   let x = read_int () in  
3 |   let y = read_int () in  
4 |   print_int (add x y);  
5 |   print_newline ()
```

```
1 | val simple_main : unit -> unit = <fun>
```

Functions: curried by default

Functions are **curried** (unlike SML functions).

`add x y` actually is `(add(x))(y)`

`add` can be partially applied, as `add x`.

```
1 | let add1 = add 1 ;;
```

```
2 |
```

```
3 | add1 2
```

```
1 | - : int = 3
```

Recursivity i

Recursive functions are explicitly qualified by the **rec** keyword.

```
1 (** [foldl f v l] is f ... (f (f v a0) a1) ... an)
2  ** assuming [l] is [a0; a1; ...; an].
3  ** [foldl] is sometimes called reduce ,*)
4  let rec foldl f acc = function
5    | [] -> acc
6    | x :: xs -> foldl f (f acc x) xs ;;

1 val foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

1 foldl (+) 0 [1;2;3;4] ;;

1 - : int = 10
```

Recursivity ii

```
1 (** [( |-> ) n m] is an infix operator computing
2    ** the list of elements from [n] included to [m] included. *)
3    let ( |-> ) lo hi =
4        let rec loop acc n =
5            if n > hi then List.rev acc
6            else loop (n :: acc) (n + 1) in
7        loop [] lo
8    ;;

1 val ( |-> ) : int -> int -> int list = <fun>

1 | 1 |-> 10 ;;

1 | - : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

Corollary

Functions are **not recursive** by default.

```
1 (* This second declaration hides the first. *)
2 let (|->) lo hi =
3   assert (hi >= lo);
4   lo |-> hi

1 val (|->) : int -> int -> int list = <fun>
```

Functions are first-class citizens i

```
1 (** [max cmp l] computes the maximum element of a list [l] provided a [cmp]
2  ** function conforming to the following specification:
3  ** - cmp x y = 0 if x is equivalent to y
4  ** - cmp x y > 0 if x is bigger than y
5  ** - cmp x y < 0 if x is smaller than y **)
6  let max cmp l =
7    let rec loop vmax = function
8      | [] -> vmax
9      | x :: xs ->
10         let vmax' =
11           match vmax with
12             | None -> Some x
13             | Some y -> if cmp x y > 0 then Some x else vmax in
14         loop vmax' xs
15    in loop None l
16 ;;

1 val max : ('a -> 'a -> int) -> 'a list -> 'a option = <fun>
```

Functions are first-class citizens ii

```
1 | max Stdlib.compare [1; 2; 3;] ;;
```

```
1 | - : int option = Some 3
```

```
1 | (* We just hid [Stdlib.max] behind another definition !*)
```

```
2 | Stdlib.max ;;
```

```
1 | - : 'a -> 'a -> 'a = <fun>
```


Evaluation is strict

Laziness (call-by-name / call-by-need) is not the default evaluation mode.

OCaml is said to be **strict** (call-by-value).

```
1 let double x = print_int x; 2 * x ;;
2
3 (* We forgot to use x ... *)
4 let dadd _x y =
5   let x' = double y
6   and y' = double y in
7   (* Infix operators are prefixed ones that are treated specially
8      by the parser. Have fun and create your owns. *)
9   ( + ) x' y' ;;
10
11 dadd (double 1) (double 2) ;;
1
1 2144- : int = 16
```

... well except for binary Boolean operators – of course ;-).

Evaluation Oddity

Evaluation order for function arguments is **unspecified**.

It is usually right-to-left, as exemplified by the snippet below.

```
1| add (print_string "foo!"; 1) (print_string "bar!"; 2) ;;
```

```
1| bar!foo!- : int = 3
```

```
1| ( || ) (print_string "foo!"; false) (print_string "bar!"; true) ;;
```

```
1| foo!bar!- : bool = true
```

Grouping information : Tuples

```
1 | let a = 1, 2 in  
2 | let x, y = a in  
3 | x + y  
4 | ;;
```

```
1 | - : int = 3
```

can also be written with (..) as

```
1 | let a = (1, 2) in  
2 | let (x, y) = a in  
3 | x + y  
4 | ;;
```

```
1 | - : int = 3
```

Everything is a pattern

```
1 | let create x y z = x, y, z
2 |
3 | (* FP arithmetic operations have a dedicated syntax *)
4 | let square a = a *. a
5 |
6 | let dist (x1, y1, z1) p =
7 |   let x2, y2, z2 = p in
8 |   let xdiff = x2 -. x1
9 |   and ydiff = y2 -. y1
10 |  and zdiff = z2 -. z1 in
11 |   square xdiff +. square ydiff +. square zdiff |> sqrt

1 | val dist : float * float * float -> float * float * float -> float = <fun>
```

dist: another version

```
1 let dist p1 p2 =  
2   match p1, p2 with  
3   (* The | can also be used as a separator instead of as a starting  
4     annotation. *)  
5   | (x1, y1, z1), (x2, y2, z2) ->  
6     let xdiff = x2 -. x1  
7     and ydiff = y2 -. y1  
8     and zdiff = z2 -. z1 in  
9     sqrt @@ square xdiff +. square ydiff +. square zdiff  
  
1  val dist : float * float * float -> float * float * float -> float = <fun>
```

Grouping information : Records (aka named tuples)

```
1 type point_2d = { x : float; y: float; } ;;
2
3 (* C-like . notations for field access *)
4 let dist p1 p2 =
5   let xdiff = p1.x -. p2.x
6   and ydiff = p1.y -. p2.y in
7   sqrt (xdiff *. xdiff +. ydiff *. ydiff)

1 val dist : point_2d -> point_2d -> float = <fun>

1 (* Using pattern-matching *)
2 let dist p1 p2 =
3   match p1, p2 with
4   | { x; y; }, { x = x'; y = y'; } ->
5     let xdiff = x -. x'
6     and ydiff = y -. y' in
7     sqrt (xdiff *. xdiff +. ydiff *. ydiff)

1 val dist : point_2d -> point_2d -> float = <fun>
```

Building / destructing records

```
1 (* Record can be built/destroyed using a shortcut notation.  
2    [let create x y = { x; y; }] is a shortcut for  
3    [let create x y = { x = x; y = y; }].  
4  
5    Choose your field names wisely and unleash your inner procrastinator !  
6 *)  
7 let create x y = { x; y; }  
8  
9 let of_int myx myy = { x = float myx; y = float myy; }  
  
1 val create : float -> float -> point_2d = <fun>  
2 val of_int : int -> int -> point_2d = <fun>
```

Records are limited to $2^{22} - 1$ fields (aka max Array size)

ADT & pattern matching

Exhaustiveness and fragility of pattern-matchings are reported by default.

```
1 type prop = (* inductively defined types do not need a rec keyword *)  
2   | Pcst of bool  
3   | Pvar of string  
4   | Pand of prop * prop  
5   | Por of prop * prop  
6   | Pnot of prop
```


Lack of exhaustiveness

```
1 let free_variables =  
2   (* The pattern matching in [loop] is well-typed but not exhaustive *)  
3   let rec loop vars = function  
4     | Pvar s -> if List.mem s vars then vars else s :: vars  
5     | Pand (p1, p2) ->  
6       let vars' = loop vars p1 in  
7       loop vars' p2  
8   in loop []
```

1 Line 3, characters 22-169:

```
2 3 | .....function  
3 4 |     | Pvar s -> if List.mem s vars then vars else s :: vars  
4 5 |     | Pand (p1, p2) ->  
5 6 |         let vars' = loop vars p1 in  
6 7 |         loop vars' p2
```

7 Warning 8: this pattern-matching is not exhaustive.

8 Here is an example of a case that is not matched:

```
9 (Pcst _|Por (_, _)|Pnot _)
```

```
10 val free_variables : prop -> string list = <fun>
```

Fragility

```
1 let free_variables =
2   (* Now it is exhaustive, but ... fragile *)
3   let rec loop vars = function
4     | Pvar s -> if List.mem s vars then vars else s :: vars
5     | Pand (p1, p2) ->
6       let vars' = loop vars p1 in loop vars' p2
7     | Por (p1, p2) ->
8       let vars' = loop vars p1 in loop vars' p2
9     | Pnot p -> loop vars p
10    (* fragile pattern-matching below.
11     * if a constructor is added, it is matched *)
12    | _ -> vars
13  in loop []

1 val free_variables : prop -> string list = <fun>
```

Or-patterns

The compact solution for this function includes or-patterns.

```
1 let free_variables =  
2   let rec loop vars = function  
3     | Pvar s -> if List.mem s vars then vars else s :: vars  
4     | Pand (p1, p2)  
5     | Por (p1, p2) -> (* 'or' pattern *)  
6       let vars' = loop vars p1 in loop vars' p2  
7     | Pnot p -> loop vars p  
8     | Pcst _ -> vars (* non-fragile pattern-matching *)  
9     (* When later adding [Bxor] constructor, the  
10      * compiler will show me where pattern-matching  
11      * is not exhaustive. *)  
12   in loop []  
  
1 val free_variables : prop -> string list = <fun>
```

For (way) more: [8]

Labels (aka named arguments)

Named arguments are mainly used for 2 reasons:

- name-based disambiguation of same type parameters

```
1 | type 'a interval = { lo : 'a; hi : 'a } ;;  
2 | let create ~lo ~hi = { lo; hi; } ;;  
  
1 | val create : lo:'a -> hi:'a -> 'a interval = <fun>
```

- homogeneous naming betting on programmers' procrastination

```
1 | (* Which version would you rather write? *)  
2 | let lo = 1 and hi = 2 in create ~lo ~hi ;;  
3 |  
4 | let lbd = 12 and ubd = 15 in create ~lo:lbd ~hi:ubd ;;  
  
1 | - : int interval = {lo = 12; hi = 15}
```

Optional arguments i

A special case of named arguments is optional arguments

```
1 (* Reusing type interval *)
2 let create ?(lo=0) hi = { lo; hi; } ;;
3
4 create 2 ;;

1 - : int interval = {lo = 0; hi = 2}

1 let create ?(lo=0) ~hi () = { lo; hi; } ;;
2
3 let ival = create ~hi:2 ();;
4
5 (* The use of partial arguments complicate partial applications. *)
6 let pp_ival ?(pre="(") ?(post="") ?(sep=",") ppf { lo; hi; } =
7   Format.fprintf ppf "@[<h>%s%d%s%d%s@" pre lo sep hi post ;;
```

Optional arguments ii

```
1 val pp_ival :
2   ?pre:string ->
3   ?post:string -> ?sep:string -> Format.formatter -> int interval -> unit =
4   <fun>

1 (* The following does not type *)
2 Format.printf "%a@." pp_ival ival ;;

1 Line 2, characters 21-28:
2 2 | Format.printf "%a@." pp_ival ival ;;;;
3                                     ^^^^^^^
4 Error: This expression has type
5     ?pre:string ->
6     ?post:string -> ?sep:string -> Format.formatter -> interval -> unit
7 but an expression was expected of type Format.formatter -> 'a -> unit
```

Optional arguments iii

```
1 (* You need to create another function *)
2 Format.printf "%a@." (fun ppf ival -> pp_ival ppf ival) ival ;;
3
4 (* The following does work though *)
5 let pp_ival2 ppf = pp_ival ppf ;;
6 Format.printf "%a@." pp_ival2 ival ;;

1 (0,2)
2 - : unit = ()
```

Optional arguments **are** option types i

```
1 type ('a, 'b) return = {
2     value : 'a;
3     explanation : 'b option;
4 }

1 (* Optional arguments of a type ['a] are really ['a option] types and ce be
2  * used that way in the body of the function *)
3 let create_return_value ?explanation value =
4     { value; explanation; }
5
6 (* Now if you have a default value [v], [Some v] needs to be used. *)
7 let create_defaulted_return_value ?(explanation="message") value =
8     { value; explanation = Some explanation; }

1 val create_return_value : ?explanation:'a -> 'b -> ('b, 'a) return = <fun>
2 val create_defaulted_return_value :
3     ?explanation:string -> 'a -> ('a, string) return = <fun>
```


Optional arguments **are** option types ii

```
1 (* The construction below does not type. *)  
2 let create_defaulted_return_value ?(explanation="message") value =  
3   { value; explanation; }
```

Using named arguments in practice

A commonly used recipe to construct function signatures which include named arguments is:

1. Put your optional arguments (?)
2. Put your named arguments (~)
3. Put the rest of your arguments

```
1| val create : ?lo:int -> hi:int -> unit -> int interval
```

Type-directed record disambiguation

```
1 type fi_pair = { x : float; y : int; }  
2  
3 (* Shadowing x and y *)  
4 type if_pair = { x : int; y : float; }  
5  
6 let addall (v1:fi_pair) v2 =  
7   let xsum = truncate v1.x + v2.x in  
8   let ysum = v1.y + truncate v2.y in  
9   xsum + ysum  
  
1 type fi_pair = { x : float; y : int; }  
2 type if_pair = { x : int; y : float; }  
3 val addall : fi_pair -> if_pair -> int = <fun>
```

See [15, 14]

Imperative programming

It's ok to be impure

A bunch of OCaml primitive constructs are imperative.

- ref
- mutable field in records
- arrays
- hashtables
- bytes (aka strings before 4.02)
- stacks, queues,

The unit type

The type of sequence elements (think 'C statement') is `unit`, which is inhabited by a single value `()`.

```
1 | let fact n =  
2 |   let res = ref 1 in  
3 |   for j = 2 to n do  
4 |     res := !res * j; (* this assignment has type unit *)  
5 |   done; (* The for loop too ! *)  
6 |   !res  
  
1 | val fact : int -> int = <fun>
```

What's in a reference?

```
1 let x = ref 1
2
3 let y : int Stdlib.ref = { contents = 12 }
4
5 type 'a ref = { mutable contents : 'a }

1 val x : int Stdlib.ref = {Stdlib.contents = 13}
2 val y : int Stdlib.ref = {Stdlib.contents = 14}
3 type 'a ref = { mutable contents : 'a; }

1 let _ = x := 13; y := 14 ;;
2
3 (!x, !y) ;;

1 - : int * int = (13, 14)
```

Typical OCaml code

Typical code mixes and matches functional and imperative features, usually for efficiency reasons.

You will not be castigated for doing so.

Cardinal

```
1 type 'a set = 'a list
2
3 let cardinal (l: 'a set) =
4   let h = Hashtbl.create 7 in
5   let rec loop = function
6     | x :: xs ->
7       Hashtbl.add h x (); (* Hashtbl.replace may be better here *)
8       loop xs
9     | [] ->
10      Hashtbl.length h
11   in loop l
```

```
1 type 'a set = 'a list
2 val cardinal : 'a set -> int = <fun>
```

String concatenation

```
1 (* Concatenating the elements of a string list.
2  * Clearly not thread safe. *)
3 let concat =
4   (* An OCaml [Buffer.t] is similar to a Java [StringBuffer].
5    * It is a self-growing array of bytes. *)
6   let b = Buffer.create 1024 in
7   fun ~sep l ->
8     Buffer.reset b;      (* cleanup any previously written contents *)
9     List.iter (fun s -> Buffer.add_string b s;
10                  Buffer.add_string b sep;) l;
11   Buffer.contents b

1 | val concat : sep:string -> string list -> string = <fun>
```

Trivia :: Hashtables

OCaml hashtables have an interesting property

`Hashtbl.add` does not remove old values!

```
1 let h = Hashtbl.create 7 ;;
2 Hashtbl.add h 1 2 ;;
3 Hashtbl.add h 1 3 ;;
4 Hashtbl.iter (fun k v -> Format.printf "%d -> %d@." k v) h ;;
```

```
1 1 -> 3
2 1 -> 2
3 - : unit = ()
```

Use `Hashtbl.replace` if you want substitution.

; Pitfall(s) i

```
1 (* This function is syntactically incorrect *)
2 let test_and_print =
3   let count_success = ref 0 in
4   fun secret ->
5     if secret = "you will never guess that" then
6       (* Uh oh *)
7       incr count_success; Format.printf "Success"
8     else Format.printf "Failure"
```

```
1 Line 8, characters 2-6:
2 8 |   else Format.printf "Failure";;
3   |   ^^^^
4 Error: Syntax error
```

; Pitfall(s) ii

```
1 (* This is correct *)
2 let test_and_print =
3   let count_success = ref 0 in
4   fun secret ->
5     if secret = "you will never guess that" then begin (* or ( *)
6       incr count_success; Format.printf "Success"
7     end (* or ) *)
8   else Format.printf "Failure"

1 val test_and_print : string -> unit = <fun>
```

Exceptions

Exceptions are **open** algebraic data types with a dedicated construct exception

```
1 exception Empty_list
2
3 let nth i l =
4   assert (i >= 0);
5   let rec aux j = function
6     | [] ->
7       raise Empty_list
8     | x :: xs ->
9       if j = 0 then x
10      else aux (j - 1) xs
11   in aux i l

1 exception Empty_list
2 val nth : int -> 'a list -> 'a = <fun>
```

Local exceptions

```
1 (** [find p a] returns
2  ** - [None] if no element of [a] verifies predicate [p]
3  ** - [Some e] otherwise where [e] is the first element of [a] s.t.
4  **   [p e = true ]
5  **)
6 let find (type a) (p:a -> bool) (arr:a array) =
7   let exception Found of a in
8   match Array.iter (fun e -> if p e then raise (Found e)) arr with
9   | () -> None
10  | exception (Found elt) -> Some elt

1 | val find : ('a -> bool) -> 'a array -> 'a option = <fun>
```

More advanced topics

What's a module?

The moment you have 2 files, you will be manipulating **modules**.

A module is an abstraction barrier to bundle together related definitions and functionalities. In particular, it defines a namespace.

A file defines a module: Both files `File` and `file` define module `File`.

Inside a module, one can define other modules, **of course**.

Modules can be recursive.

Interface (**mli**) & implementation (**m1**)

OCaml programmers detach API interfaces (documentation, typing, ...) from their implementation in separate files

Usage: Do **not** type every and all functions coded, just the exported ones and those whose type cannot be correctly inferred.

So for a given file-based module

- `mli` files contain interfaces (aka type signatures)
- `m1` files contain implementations

For “programmatic” modules, you will use `module` type to abstract functionalities/traits and `module` for implementing said modules.

Implementation

```

1 type 'a set = 'a list
2 let empty = []
3 let singleton e = [e]
4 let mem = List.mem
5 let add e s = if mem e s then s else e :: s
6 let union s1 s2 = List.fold_left (fun s e -> add e s) s1 s2
7 let intersection s1 s2 =
8   List.fold_left (fun s e -> if mem e s1 then e :: s else s) [] s2

```

Interface

```

1 type 'a set
2 val empty: 'a set
3 val singleton : 'a -> 'a set
4 val mem: 'a -> 'a set -> bool
5 val add: 'a -> 'a set -> 'a set
6 val union: 'a set -> 'a set -> 'a set
7 val intersection: 'a set -> 'a set -> 'a set

```

Levels of type abstraction

Modules help to delimit abstraction barriers and one can choose the desired level of abstraction.

Open types can be directly constructed & destructed

```
1 module Interval_concrete = struct
2   type t =
3     | Ival of { lo : int; hi : int; } (* here's an inline record *)
4     | Top
5
6   let top = Top
7
8   let ival ~lo ~hi =
9     assert (lo <= hi);
10    if lo = min_int && hi = max_int then top
11    else Ival {lo; hi;}
12 end
```

Open visibility ii

```
1 open Interval_concrete
2
3 (* Pattern-matching is ok *)
4 let size = function
5   | Ival {lo; hi; } -> float @@ hi - lo + 1
6   | Top -> infinity
7
8 (* This is authorized. *)
9 let interval = Top
10
11 (* This is too
12  * the [top] function is part of the module signature *)
13 let top2 = top
```

Private visibility i

Private types can be:

- constructed only by predefined module-local functions,
- destructured by usual pattern matching.

```
1 module type IPRIVATE = sig
2   type t = private
3       | Ival of { lo : int; hi: int; }
4       | Top
5
6   val ival : lo:int -> hi:int -> t
7   val top  : t
8 end
```

Private visibility ii

```
1 module Interval_private : IPRIVATE = Interval_concrete
2
3 open Interval_private
4
5 (* Pattern-matching is ok on private types *)
6 let size = function
7   | Ival {lo; hi; } -> float @@ hi - lo + 1
8   | Top -> infinity
9
10 (* This is ok * the [top] function is part of
11  * the [IPRIVATE] module signature *)
12 let top2 = top
13
14 module Interval_private : IPRIVATE
15 val size : Interval_private.t -> float = <fun>
16 val top2 : Interval_private.t = Interval_private.Top
```


Private visibility iii

```
1 (* This is not authorized. *)  
2 let interval = Top
```

```
1 Line 2, characters 15-18:
```

```
2 2 | let interval = Top;;  
3                      ^^^
```

```
4 Error: Cannot create values of the private type Interval_private.t
```

Abstract visibility i

Abstract types can be constructed & destructed only be predefined functions

```
1 module type IABSTRACT = sig
2   type t (* opaque to the outside world *)
3   val ival : lo:int -> hi:int -> t ;;
4   val top : t
5
6   (** Accessors needs to be in the interface now *)
7   val is_top : t -> bool
8
9   (** Fails if value is [!top] *)
10  val lo : t -> int
11  val hi : t -> int
12 end
```

Abstract visibility ii

```
1 module Interval_abstract : IABSTRACT = struct
2   include Interval_concrete
3
4   let is_top = function Top -> true
5   | Ival _ -> false
6
7   let lo = function
8   | Ival i -> i.lo
9   | Top -> assert false
10
11  let hi = function
12  | Ival i -> i.hi
13  | Top -> assert false
14 end
```

Abstract visibility iii

```
1 open Interval_abstract
2
3 (* Pattern-matching does not work anymore *)
4 let size ival =
5   if is_top ival then infinity
6   else float @@ hi ival - lo ival + 1
7
8 (* This is ok for the [top] function is part
9   of the module signature *)
10 let top2 = top
11
12 val size : Interval_abstract.t -> float = <fun>
13 val top2 : Interval_abstract.t = <abstr>
14
15 (* This is still not authorized. *)
16 let interval = Top
```

Abstract visibility iv

```
1 | Line 2, characters 15-18:  
2 | 2 | let interval = Top;;  
3 |                      ^^^  
4 | Error: Cannot create values of the private type Interval_private.t
```

Genericity: let-polymorphism

```
1 | let rec length = function
2 |   | [] -> 0
3 |   | _ :: l' -> 1 + length l'
4 | ;;

1 | val length : 'a list -> int = <fun>
```

Genericity: Functors i

The standard library has functors for sets, maps (tree-based persistent dictionaries) and hashtables.

```
1 module type PRINTABLE = sig
2   type t
3   val pp: Format.formatter -> t -> unit
4 end
5
6 module List_printer(X:PRINTABLE) = struct
7   let pp_list
8     ?(pre=(fun ppf () -> Format.pp_print_string ppf "["))
9     ?(post=(fun ppf () -> Format.pp_print_string ppf "]"))
10    ?(sep=(fun ppf () -> Format.fprintf ppf ";@ "))
11    ppf l =
12    let open Format in
13    let rec loop = function
14      | [] -> post ppf ()
```

Genericity: Functors ii

```
15 |         | e :: es ->
16 |         begin
17 |             X.pp ppf e;
18 |             sep ppf ();
19 |             loop es
20 |         end
21 |     in pre ppf (); loop l
22 | end

1 | module Int_list_pp =
2 |     List_printer(struct type t = int let pp = Format.pp_print_int end)
3 |
4 | let pp_ilst ppf l = Int_list_pp.pp_list ppf l ;;
5 | pp_ilst Format.std_formatter [1;2;3]

1 | [1; 2; 3;
2 | ]- : unit = ()
```


Genericity: Functors iii

```
1 module String_list_pp =  
2   List_printer(  
3     struct  
4       type t = string  
5       let pp = Format.pp_print_string  
6     end)  
7  
8   let pp_slist = fun ppf l -> String_list_pp.pp_list ppf l;;  
9   Format.printf "@[<h>%a@]" pp_slist ["foo"; "bar"; "bar"];;  
  
1  [foo; bar; bar; ]- : unit = ()
```

First-class modules

Modules can also be used as “first-class” values.

```
1 module type COMPARABLE = sig
2   type t
3   val compare : t -> t -> int
4 end
5
6 let lmax (type a) (module M:COMPARABLE with type t = a) (l:a list) =
7   let rec aux vmax l =
8     match l with
9     | [] -> vmax
10    | x :: xs ->
11      let vmax' =
12        match vmax with
13        | None -> Some x
14        | Some v -> if M.compare x v > 0 then Some x else vmax in
15      aux vmax' xs
16   in aux None l
17
18 module Int = struct type t = int let compare = Stdlib.compare end ;;
```

Using first-class module

```
1 | lmax (module Int) [1;2;3;] ;;  
1 | - : Int.t option = Some 3  
  
1 | (* Module [String] is part of the standard library *)  
2 | lmax (module String) ["foo"; "bar"; "baz";] ;;  
  
1 | - : String.t option = Some "foo"
```

More involved example i

```
1 type ('var, 'cst, 'bop, 'uop) expr =  
2   | Var of 'var  
3   | Cst of 'cst  
4   | Bop of 'bop *  
5       ('var, 'cst, 'bop, 'uop) expr *  
6       ('var, 'cst, 'bop, 'uop) expr  
7   | Uop of 'uop * ('var, 'cst, 'bop, 'uop) expr  
8  
9 module type EXPR = sig  
10  type var  
11  type uop  
12  type cst  
13  type bop  
14 end
```

Defining an EXPR module

```
1 module Bool = struct
2   type bop =
3     | Band
4     | Bor
5     | Bxor
6
7   type uop = Bnot
8
9   type var = string
10
11  type cst = bool
12 end
```

Generic free variable computation

```
1 let free_variables (type a b c d)
2   (module M:EXPR with type var = a and type cst = b and
3     type bop = c and type uop = d)
4   (e:(a,b,c,d) expr) : a list =
5   let module S =
6     Set.Make(struct type t = M.var let compare = Stdlib.compare end) in
7   let rec loop (set:S.t) = function
8     | Var v -> S.add v set
9     | Cst _ -> set
10    | Bop (_, e1, e2) -> S.union (loop set e1) (loop S.empty e2)
11    | Uop (_, e) -> loop set e
12  in
13  let set = loop S.empty e in
14  S.fold List.cons set []
15 ;;

1 val free_variables :
2   (module EXPR with type bop = 'a and type cst = 'b and type uop = 'c and type var
3     ('d, 'b, 'a, 'c) expr -> 'd list = <fun>
4   free_variables (module Bool) (Var "foo") ;;
5
6 - : Bool.var list = ["foo"]
```

Monadic style programming

The following type has been making a comeback.

```
1 | type ('a, 'b) result =  
2 |   | Ok of 'a  
3 |   | Error of 'b
```

and with it, monadic-style programming

There is no dedicated notation (no `do`) for working inside monads.

One usually directly luses the `M.bind` function of monad `M` or,
define an infix operator (`>>=`)

Option type, monadic style

```
1 let (>=>) = Option.bind
2
3 let hd = function
4   | [] -> None
5   | x :: _ -> Some x
6
7 let sum_heads l1 l2 =
8   hd l1 >=>
9     fun v1 -> hd l2 >=>
10      fun v2 -> v1 + v2 |> Option.some
11
12 val ( >=> ) : 'a option -> ('a -> 'b option) -> 'b option = <fun>
13 val hd : 'a list -> 'a option = <fun>
14 val sum_heads : int list -> int list -> int option = <fun>
```


Generalized Abstract Data Types are available in OCaml since version 4.00 ([13, 10]).

They are sparsely used but can be pretty useful.

```
1 type _ bop =  
2   | Add : int bop  
3   | Mul : int bop  
4   | Div : int bop  
5   | Bor : bool bop  
6   | Band : bool bop  
7  
8 type _ uop =  
9   | UMin : int uop  
10  | Bnot : bool uop  
11  
12 type comparison = Eq | Gt
```

GADTs ii

```
1 type _ typ =
2   | Int  : int -> int typ
3   | Bool : bool -> bool typ
4   | Ite  : bool typ * 'a typ * 'a typ -> 'a typ
5   | Bin  : 'a bop * 'a typ * 'a typ -> 'a typ
6   | Un   : 'a uop * 'a typ -> 'a typ
7   | Cmp  : comparison * 'a typ * 'a typ -> bool typ

1 let term = Ite (Cmp (Eq, Int 3, Int 4), Int 12, Int 11)
2
3 let term2 =
4   Ite (Cmp (Eq, Int 3, Un (UMin, Int 2)),
5       Bool true, Un (Bnot, Bool true)) ;;

1 val term : int typ = Ite (Cmp (Eq, Int 3, Int 4), Int 12, Int 11)
2 val term2 : bool typ =
3   Ite (Cmp (Eq, Int 3, Un (UMin, Int 2)), Bool true, Un (Bnot, Bool true))
```

```

1 let eval_bop: type a. a bop -> a -> a -> a = function
2   | Add -> ( + )
3   | Mul -> ( * )
4   | Div -> ( / )
5   | Bor -> ( || )
6   | Band -> ( && )
7
8 let eval_cmp = function Eq -> ( = ) | Gt -> ( > ) ;;
9
10 let rec eval: type a. a typ -> a = function
11   | Int n -> n
12   | Bool b -> b
13   | Ite (b, csq, alt) -> if eval b then eval csq else eval alt
14   | Bin (op, e1, e2) -> eval_bop op (eval e1) (eval e2)
15   | Un (UMin, e) -> - (eval e)
16   | Un (Bnot, e) -> not (eval e)
17   | Cmp (op, e1, e2) -> (eval_cmp op) (eval e1) (eval e2)
18   ;;

```

```
1 (* Ite (Cmp (Eq, Int 3, Int 4), Int 12, Int 11) *)  
2 eval term ;;
```

```
1 - : int = 11
```

```
1 (* let term2 =  
2   *   Ite (Cmp (Eq, Int 3, Un (UMin, Int 2)),  
3   *       Bool true, Un (Bnot, Bool true)) ;; *)  
4  
5 eval term2 ;;
```

```
1 - : bool = false
```

Death by GADTs

With great expressiveness may come great unreadability ;-)

```
1 (* in stdlib/camlinternalFormatBasics.ml *)
2 and ('a1, 'b1, 'c1, 'd1, 'e1, 'f1,
3      'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel =
4   | Char_ty : (* %C *)
5     ('a1, 'b1, 'c1, 'd1, 'e1, 'f1,
6      'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel ->
7     (char -> 'a1, 'b1, 'c1, 'd1, 'e1, 'f1,
8      char -> 'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel
9   | String_ty : (* %S *)
10    ('a1, 'b1, 'c1, 'd1, 'e1, 'f1,
11     'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel ->
12    (string -> 'a1, 'b1, 'c1, 'd1, 'e1, 'f1,
13     string -> 'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel

1 (* same file *)
2 let rec erase_rel : type a b c d e f g h i j k l .
3   (a, b, c, d, e, f, g, h, i, j, k, l) fmtty_rel ->
4   (a, b, c, d, e, f) fmtty
```

The Format module is a pretty-printing facility available in the standard library with a Printf-like string format.

Format structures outputs using **boxes** and **break hints**.

3 salient elements to think about

- `Format.fprintf`
- the formatter abstraction (one level above `output_channel`)
- `%a` to chain pretty printers

Format example

```
1 module E = struct
2   type t =
3     | Int of int
4     | Add of t * t
5
6   let rec pp_expr ppf = function
7     | Int n -> Format.fprintf ppf "%02d" n
8     | Add (e1, e2) ->
9       Format.fprintf ppf "%a +@ %a" pp_expr e1 pp_expr e2
10 end
1
1 let () =
2   let open E in
3   List.fold_left (fun e n -> Add (Int n, e)) (Int 0) (1 |-> 20)
4   |> Format.printf "@[<hov>%a@]@." pp_expr
1 20 + 19 + 18 + 17 + 16 + 15 + 14 + 13 + 12 + 11 + 10 + 09 + 08 + 07 + 06 +
2 05 + 04 + 03 + 02 + 01 + 00
```

For more: [2]

Polymorphic variants i

```
1 type const = [ `True | `False ]
2
3 (* See e.g., https://en.wikipedia.org/wiki/NAND\_logic *)
4 let rec nandify = function
5   | #const as b -> b
6   | `Bnot b ->
7     let b' = nandify b in `Bband (b', b')
8   | `Bband (b1, b2) ->
9     let b1 = nandify b1 and b2 = nandify b2 in
10    `Bband (`Bband (b1, b2), `Bband (b1, b2))
11   | `Bband (b1, b2) ->
12    `Bband(nandify b1, nandify b2)
13   | `Bor (b1, b2) ->
14     let b1 = nandify b1 and b2 = nandify b2 in
15    `Bband (`Bband (b1, b1), `Bband (b2, b2))
```


Polymorphic variants ii

```
1 type const = [ `False | `True ]
2 val nandify :
3   ([< `Band of 'a * 'a
4     | `Bband of 'a * 'a
5     | `Bnot of 'a
6     | `Bor of 'a * 'a
7     | `False
8     | `True ]
9   as 'a) ->
10  ([> `Bband of 'b * 'b | `False | `True ] as 'b) = <fun>
```

See [16, 3, 4]

Things that I did not talk about

- Low-level representation [5]
- Objects
- Laziness (lazy keyword, streams, ..., Seq) [9]
- GC [1, 6]
- PPX syntax extensions (deriving, sexp, ...)
- FFI
- Ecosystem
 - parser generator: ocamlyacc, Menhir [11, 12]
 - libraries
 - ...

Conclusion

OCaml

A pragmatic functional language

Try it at <https://ocaml.org/>

With OCaml, you're not learning the computer programming of the last 10 years, you're learning the programming of the 10 coming years.

–

Sylvain Conchon

<https://www.ocamlpro.com/2020/06/05/interview-sylvain-conchon-cso-on-formal-methods/>

Questions?



https://github.com/rbonichon/skillsmatter_20200625

References

- [1] T. Blanc. OCaml new best fit GC. 2020. URL <https://www.ocamlpro.com/2020/03/23/ocaml-new-best-fit-garbage-collector/>.
- [2] R. Bonichon and P. Weis. Format unraveled. *JFLA*, 2018. URL <https://hal.archives-ouvertes.fr/hal-01503081/file/format-unraveled.pdf>.
- [3] J. Garrigue. Programming with polymorphic variants. *ML Workshop*, 1998. URL <https://www.math.nagoya-u.ac.jp/~garrigue/papers/variants.pdf>.
- [4] J. Garrigue. Code reuse through polymorphic variants. *ML Workshop*, 2000. URL <https://www.math.nagoya-u.ac.jp/~garrigue/papers/variant-reuse.pdf>.
- [5] J. Hickey, A. Madhavapeddy, and Y. Minsky. Memory representation of values. 2014. URL <http://dev.realworldocaml.org/runtime-memory-layout.html>.
- [6] J. Hickey, A. Madhavapeddy, and Y. Minsky. Understanding

the garbage collector. 2020. URL

<http://dev.realworldocaml.org/garbage-collector.html>.

[7] C. Kreitz and V. Rahli. Introduction to classic ml. 2011. URL

<http://www.nuprl.org/software/eventml/KreitzandRahli-ClassicML.pdf>.

[8] F. Le Fessant and L. Maranget. Optimizing pattern matching. *SIGPLAN Not.*, 36(10):26–37, Oct. 2001. ISSN 0362–1340. doi: 10.1145/507669.507641. URL <https://doi.org/10.1145/507669.507641>.

[9] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. RémyP, and J. Vouillon. The ocaml system release 4.10. 2020. URL

<https://caml.inria.fr/pub/docs/manual-ocaml/>.

[10] A. Madhavapeddy and J. Yallop. Programming with GADTS. 2014. URL <https://www.cl.cam.ac.uk/teaching/1415/L28/gadts.pdf>.

[11] F. Pottier. Reachability and error diagnosis in LR(1) parsers. In *International Conference on Compiler Construction (CC)*, pages 88–98, Mar. 2016. doi:

<http://dx.doi.org/10.1145/2892208.2892224>. URL

<http://gallium.inria.fr/~fpottier/publis/fpottier-reachability-cc2016.pdf>.

- [12] F. Pottier and Y. Régis-Gianas. Towards efficient, typed LR parsers. In *ACM Workshop on ML (ML)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 155–180, Mar. 2006. URL

<http://gallium.inria.fr/~fpottier/publis/fpottier-regis-gianas-typed-lr.pdf>.

- [13] F. Pottier and Y. Régis-Gianas. Stratified type inference for generalized algebraic data types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 232–244, Charleston, South Carolina, Jan. 2006. doi: <http://doi.acm.org/10.1145/1111037.1111058>. URL

<http://gallium.inria.fr/~fpottier/publis/pottier-regis-gianas-popl06.pdf>.

- [14] G. Scherer. Resolving field names. 2012. URL

<http://gallium.inria.fr/~scherer/gagallium/resolving-field-names/>.

- [15] G. Scherer. Resolving field names (part 2). 2012. URL <http://gallium.inria.fr/~scherer/gagallium/resolving-field-names-2/>.
- [16] B. Yakobowski. Le caractère ' à la rescousse: Factorisation et réutilisation de code grâce aux variants polymorphes. In *JFLA*, 2008. URL <http://www.yakobowski.org/publis/2008/jfla08.pdf>.