Introduction to OCaml

Richard Bonichon

20200625

TWEAG Software Innovation Lab

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`

Users

- 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

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

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

```
let simple_main () =
  let x = read_int () in
  let y = read_int () in
  print_int (add x y);
  print_newline ()
```

```
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`.

```
let add1 = add 1 ;;

add1 2
```

```
- : int = 3
```

Corollary

Functions are not recursive by default.

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

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

Functions are first-class citizens i

```
(** [max cmp l] computes the maximun element of a list [l] provided a [cmp]
 ** function conforming to the following specification:
 ** - cmp x y = 0 if x is equivalent to y
 ** - cmp x y > 0 if x is bigger than y
 ** - cmp x y < 0 if x if smaller than y **)
let max cmp l =
  let rec loop vmax = function
    | [] -> vmax
    | x :: xs ->
      let vmax' =
        match vmax with
        | None -> Some x
        | Some y -> if cmp x y > 0 then Some x else vmax in
      loop vmax' xs
  in loop None l
;;
```

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

Functions are first-class citizens ii

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

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

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

```
- : '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).

```
let double x = print_int x; 2 * x ;;

(* We forgot to use x ... *)
let dadd _x y =
  let x' = double y
  and y' = double y in
  (* Infix operators are prefixed ones that are treated specially
     by the parser. Have fun and create your owns. *)
  ( + ) x' y' ;;

dadd (double 1) (double 2) ;;
```

```
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.

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

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

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

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

Grouping information : Tuples

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

```
- : int = 3
```

can also be written with `( .. )` as

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

```
- : int = 3
```

Everything is a pattern

```ocaml
let create x y z = x, y, z

(* FP arithmetic operations have a dedicated syntax *)
let square a = a *. a

let dist (x1, y1, z1) p =
  let x2, y2, z2 = p in
  let xdiff = x2 -. x1
  and ydiff = y2 -. y1
  and zdiff = z2 -. z1 in
  square xdiff +. square ydiff +. square zdiff |> sqrt
```

```ocaml
val dist : float * float * float -> float * float * float -> float = <fun>
```

`dist`: another version

```
let dist p1 p2 =
  match p1, p2 with
  (* The | can also be used as a separator instead of as a starting
     annotation. *)
  | (x1, y1, z1), (x2, y2, z2) ->
     let xdiff = x2 -. x1
     and ydiff = y2 -. y1
     and zdiff = z2 -. z1 in
     sqrt @@ square xdiff +. square ydiff +. square zdiff
```

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

Grouping information : Records (aka named tuples)

```
type point_2d = { x : float; y: float; }  ;;

(* C-like . notations for field access *)
let dist p1 p2 =
  let xdiff = p1.x -. p2.x
  and ydiff = p1.y -. p2.y in
  sqrt (xdiff *. xdiff +. ydiff *. ydiff)
```

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

```
(* Using pattern-matching *)
let dist p1 p2 =
  match p1, p2 with
  | { x; y; }, { x = x'; y = y';} ->
     let xdiff = x -. x'
     and ydiff = y -. y' in
     sqrt (xdiff *. xdiff +. ydiff *. ydiff)
```

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

Building / destructing records

```ocaml
(* Record can be built/destructed using a shortcut notation.
   [let create x y = { x; y; }] is a shortcut for
   [let create x y = { x = x; y = y; }].

   Choose your field names wisely and unleash your inner procrastinator !
 *)
let create x y = { x; y; }

let of_int myx myy = { x = float myx; y = float myy; }
```

```ocaml
val create : float -> float -> point_2d = <fun>
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.

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

Lack of exhaustiveness

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

```
Line 3, characters 22-169:
3 | ......................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
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Pcst _|Por (_, _)|Pnot _)
val free_variables : prop -> string list = <fun>
```

Fragility

```
let free_variables =
  (* Now it is exhaustive, but ... fragile *)
  let rec loop vars = function
    | Pvar s -> if List.mem s vars then vars else s :: vars
    | Pand (p1, p2) ->
       let vars' = loop vars p1 in loop vars' p2
    | Por (p1, p2) ->
       let vars' = loop vars p1 in loop vars' p2
    | Pnot p -> loop vars p
    (* fragile pattern-matching below.
     * if a constructor is added, it is matched *)
    | _ -> vars
  in loop []
```

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

Or-patterns

The compact solution for this function includes or-patterns.

```
let free_variables =
  let rec loop vars = function
    | Pvar s -> if List.mem s vars then vars else s :: vars
    | Pand (p1, p2)
    | Por (p1, p2) -> (* 'or' pattern *)
       let vars' = loop vars p1 in loop vars' p2
    | Pnot p -> loop vars p
    | Pcst _ -> vars (* non-fragile pattern-matching *)
    (* When later adding [Bxor] constructor, the
     * compiler will show me where pattern-matching
     * is not exhaustive. *)
  in loop []
```

```
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

```
type 'a interval = { lo : 'a; hi : 'a } ;;
let create ~lo ~hi = { lo; hi; } ;;
```

```
val create : lo:'a -> hi:'a -> 'a interval = <fun>
```

- homogeneous naming betting on programmers' procrastination

```
(* Which version would you rather write? *)
let lo = 1 and hi = 2 in create ~lo ~hi  ;;

let lbd = 12 and ubd = 15 in create ~lo:lbd ~hi:ubd ;;
```

```
- : int interval = {lo = 12; hi = 15}
```

Optional arguments i

A special case of named arguments is optional arguments

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

create 2 ;;
```

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

```
let create ?(lo=0) ~hi () = { lo; hi; } ;;

let ival = create ~hi:2 ();;

(* The use of partial arguments complicate partial applications.*)
let pp_ival ?(pre="(") ?(post=")") ?(sep=",") ppf { lo; hi; } =
  Format.fprintf ppf "@[<h>%s%d%s%d%s@]" pre lo sep hi post ;;
```

Optional arguments ii

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

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

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

Optional arguments iii

```
(* You need to create another function *)
Format.printf "%a@." (fun ppf ival -> pp_ival ppf ival) ival ;;

(* The following does work though *)
let pp_ival2 ppf = pp_ival ppf ;;
Format.printf "%a@." pp_ival2 ival ;;
```

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

Optional arguments are option types i

```ocaml
type ('a, 'b) return = {
    value : 'a;
    explanation : 'b option;
  }
```

```ocaml
(* Optional arguments of a type ['a] are really ['a option] types and ce be
 * used that way in the body of the function *)
let create_return_value ?explanation value =
  { value; explanation; }

(* Now if you have a default value [v], [Some v] needs to be used. *)
let create_defaulted_return_value ?(explanation="message") value =
  { value; explanation = Some explanation; }
```

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

Optional arguments are option types ii

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

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

Type-directed record disambiguation

```
type fi_pair = { x : float; y : int; }

(* Shadowing x and y *)
type if_pair = { x : int; y : float; }

let addall (v1:fi_pair) v2 =
  let xsum = truncate v1.x + v2.x in
  let ysum = v1.y + truncate v2.y in
  xsum + ysum
```

```
type fi_pair = { x : float; y : int; }
type if_pair = { x : int; y : float; }
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 `()`.

```
let fact n =
  let res = ref 1 in
  for j = 2 to n do
    res := !res * j; (* this assignment has type unit *)
  done; (* The for loop too ! *)
  !res
```

```
val fact : int -> int = <fun>
```

What’s in a reference?

```
let x = ref 1

let y : int Stdlib.ref = { contents = 12 }

type 'a ref = { mutable contents : 'a }
```

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

```
let _ = x := 13; y := 14 ;;

(!x, !y) ;;
```

```
- : 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

```ocaml
type 'a set = 'a list

let cardinal (l:'a set) =
  let h = Hashtbl.create 7 in
  let rec loop = function
    | x :: xs ->
       Hashtbl.add h x (); (* Hashtbl.replace may be better here *)
       loop xs
    | [] ->
       Hashtbl.length h
  in loop l
```

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

String concatenation

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

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

Trivia :: Hashtables

OCaml hashtables have an interesting property

`Hashtbl.add` does not remove old values!

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

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

Use `Hashtbl.replace` if you want substitution.

; Pitfall(s) i

```ocaml
1 (* This function is syntacticlly 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"
```

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

; Pitfall(s) ii

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

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

Exceptions

Exceptions are open algebraic data types with a dedicated construct exception

```
exception Empty_list

let nth i l =
  assert (i >= 0);
  let rec aux j = function
    | [] ->
        raise Empty_list
    | x :: xs ->
        if j = 0 then x
        else aux (j - 1) xs
  in aux i l
```

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

Local exceptions

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

```
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 (`ml`)

`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)
- `ml` files contain implementations

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

Implementation

```ocaml
type 'a set = 'a list
let empty = []
let singleton e = [e]
let mem = List.mem
let add e s = if mem e s then s else e :: s
let union s1 s2 = List.fold_left (fun s e -> add e s) s1 s2
let intersection s1 s2 =
  List.fold_left (fun s e -> if mem e s1 then e :: s else s) [] s2
```

Interface

```ocaml
type 'a set
val empty: 'a set
val singleton : 'a -> 'a set
val mem: 'a -> 'a set -> bool
val add: 'a -> 'a set -> 'a set
val union: 'a set -> 'a set -> 'a set
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 visibility i

Open types can be directly constructed & destructed

```
module Interval_concrete = struct
  type t =
  | Ival of { lo : int; hi : int; } (* here's an inline record *)
  | Top

  let top = Top

  let ival ~lo ~hi =
    assert (lo <= hi);
    if lo = min_int && hi = max_int then top
    else Ival {lo; hi;}
end
```

Open visibility ii

```ocaml
open Interval_concrete

(* Pattern-matching is ok *)
let size = function
  | Ival {lo; hi; } -> float @@ hi - lo + 1
  | Top -> infinity

(* This is authorized. *)
let interval = Top

(* This is too
 * the [top] function is part of the module signature *)
let top2 = top
```

Private visibility i

Private types can be:

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

```
module type IPRIVATE = sig
  type t = private
          | Ival of { lo : int; hi: int;}
          | Top

  val ival : lo:int -> hi:int -> t
  val top : t
end
```

Private visibility ii

```
module Interval_private : IPRIVATE = Interval_concrete

open Interval_private

(* Pattern-matching is ok on private types *)
let size = function
  | Ival {lo; hi; } -> float @@ hi - lo + 1
  | Top -> infinity

(* This is ok * the [top] function is part of
 * the [IPRIVATE] module signature *)
let top2 = top
```

```
module Interval_private : IPRIVATE
val size : Interval_private.t -> float = <fun>
val top2 : Interval_private.t = Interval_private.Top
```

Private visibility iii

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

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

Abstract visibility i

Abstract types can be constructed & destructed only be predefined functions

```
module type IABSTRACT = sig
  type t (* opaque to the outside world *)
  val ival : lo:int -> hi:int -> t ;;
  val top : t

  (** Accessors needs to be in the interface now *)
  val is_top : t -> bool

  (** Fails if value is [!top] *)
  val lo : t -> int
  val hi : t -> int
end
```

Abstract visibility ii

```
module Interval_abstract : IABSTRACT = struct
  include Interval_concrete

  let is_top = function Top -> true
  | Ival _ -> false

   let lo = function
   | Ival i -> i.lo
   | Top -> assert false

   let hi = function
   | Ival i -> i.hi
   | Top -> assert false
end
```

Abstract visibility iii

```
open Interval_abstract

(* Pattern-matching does not work anymore *)
let size ival =
  if is_top ival then infinity
  else float @@ hi ival - lo ival + 1

(* This is ok for the [top] function is part
   of the module signature *)
let top2 = top
```

```
val size : Interval_abstract.t -> float = <fun>
val top2 : Interval_abstract.t = <abstr>
```

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

Abstract visibility iv

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

Genericity: let-polymorphism

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

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

Genericity: Functors i

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

```
module type PRINTABLE = sig
  type t
  val pp: Format.formatter -> t -> unit
end

module List_printer(X:PRINTABLE) = struct
  let pp_list
        ?(pre=(fun ppf () -> Format.pp_print_string ppf "["))
        ?(post=(fun ppf () -> Format.pp_print_string ppf "]"))
        ?(sep=(fun ppf () -> Format.fprintf ppf ";@ "))
        ppf l =
    let open Format in
    let rec loop = function
      | [] -> post ppf ()
```

Genericity: Functors ii

```
      | e :: es ->
        begin
          X.pp ppf e;
          sep ppf ();
          loop es
        end
    in pre ppf (); loop l
end
```

```
module Int_list_pp =
  List_printer(struct type t = int let pp = Format.pp_print_int end)

let pp_ilist ppf l = Int_list_pp.pp_list ppf l ;;
pp_ilist Format.std_formatter [1;2;3]
```

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

Genericity: Functors iii

```
module String_list_pp =
  List_printer(
      struct
        type t = string
        let pp = Format.pp_print_string
      end)

let pp_slist = fun ppf l -> String_list_pp.pp_list ppf l;;
Format.printf "@[<h>%a@]" pp_slist  ["foo"; "bar"; "bar";] ;;
```

```
[foo; bar; bar; ]- : unit = ()
```

First-class modules

Modules can also be used as "first-class" values.

```
module type COMPARABLE = sig
  type t
  val compare : t -> t -> int
end

let lmax (type a) (module M:COMPARABLE with type t = a) (l:a list) =
  let rec aux vmax l =
    match l with
    | [] -> vmax
    | x :: xs ->
       let vmax' =
         match vmax with
         | None -> Some x
         | Some v -> if M.compare x v > 0 then Some x else vmax in
       aux vmax' xs
  in aux None l

module Int = struct type t = int let compare = Stdlib.compare end ;;
```

Using first-class module

```
lmax (module Int) [1;2;3;] ;;
```

```
- : Int.t option = Some 3
```

```
(* Module [String] is part of the standard library *)
lmax (module String) ["foo"; "bar"; "baz";] ;;
```

```
- : String.t option = Some "foo"
```

More involved example i

```
type ('var,'cst,'bop,'uop) expr =
  | Var of 'var
  | Cst of 'cst
  | Bop of 'bop *
             ('var,'cst,'bop,'uop) expr *
               ('var,'cst,'bop,'uop) expr
  | Uop of 'uop * ('var,'cst,'bop,'uop) expr

module type EXPR = sig
  type var
  type uop
  type cst
  type bop
end
```

Defining an EXPR module

```
module Bool = struct
  type bop =
    | Band
    | Bor
    | Bxor

  type uop = Bnot

  type var = string

  type cst = bool
end
```

Generic free variable computation

```
let free_variables (type a b c d)
      (module M:EXPR with type var = a and type cst = b and
                             type bop = c and type uop = d)
      (e:(a,b,c,d) expr) : a list =
  let module S =
    Set.Make(struct type t = M.var let compare = Stdlib.compare end) in
  let rec loop (set:S.t) = function
    | Var v -> S.add v set
    | Cst _ -> set
    | Bop (_, e1, e2) -> S.union (loop set e1) (loop S.empty e2)
    | Uop (_, e) -> loop set e
  in
  let set = loop S.empty e in
  S.fold List.cons set []
;;
```

```
val free_variables :
  (module EXPR with type bop = 'a and type cst = 'b and type uop = 'c and type var
  ('d, 'b, 'a, 'c) expr -> 'd list = <fun>
```

```
free_variables (module Bool) (Var "foo") ;;
```

```
- : Bool.var list = ["foo"]
```

Monadic style programming

The following type has been making a comeback.

```
type ('a, 'b) result =
  | Ok of 'a
  | 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

```
let (>>=) = Option.bind

let hd = function
  | [] -> None
  | x :: _ -> Some x

let sum_heads l1 l2 =
  hd l1 >>=
    fun v1 -> hd l2 >>=
    fun v2 -> v1 + v2 |> Option.some
```

```
val ( >>= ) : 'a option -> ('a -> 'b option) -> 'b option = <fun>
val hd : 'a list -> 'a option = <fun>
val sum_heads : int list -> int list -> int option = <fun>
```

GADTs i

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

They are sparsely used but can be pretty useful.

```ocaml
type _ bop =
  | Add : int bop
  | Mul : int bop
  | Div : int bop
  | Bor : bool bop
  | Band : bool bop

type _ uop =
  | UMin : int uop
  | Bnot : bool uop

type comparison = Eq | Gt
```

GADTs ii

```
type _ typ =
  | Int  : int -> int typ
  | Bool : bool -> bool typ
  | Ite  : bool typ * 'a typ * 'a typ -> 'a typ
  | Bin  : 'a bop * 'a typ * 'a typ -> 'a typ
  | Un   : 'a uop * 'a typ -> 'a typ
  | Cmp  : comparison * 'a typ * 'a typ -> bool typ
```

```
let term = Ite (Cmp (Eq, Int 3, Int 4), Int 12, Int 11)

let term2 =
  Ite (Cmp (Eq, Int 3, Un (UMin, Int 2)),
       Bool true, Un (Bnot, Bool true)) ;;
```

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

GADTs iii

```
let eval_bop: type a. a bop -> a -> a -> a = function
  | Add -> ( + )
  | Mul -> ( * )
  | Div -> ( / )
  | Bor -> ( || )
  | Band -> ( && )

let eval_cmp = function Eq -> ( = ) | Gt -> ( > ) ;;

let rec eval: type a. a typ -> a  = function
  | Int n -> n
  | Bool b -> b
  | Ite (b, csq, alt) -> if eval b then eval csq else eval alt
  | Bin (op, e1, e2) -> eval_bop op (eval e1) (eval e2)
  | Un (UMin, e) -> - (eval e)
  | Un (Bnot, e) -> not (eval e)
  | Cmp (op, e1, e2) -> (eval_cmp op) (eval e1) (eval e2)
  ;;
```

GADTs iv

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

```
- : int = 11
```

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

eval term2 ;;
```

```
- : bool = false
```

Death by GADTs

With great expressiveness may come great unreadability ;-)

```
(* in stdlib/camlinternalFormatBasics.ml *)
and ('a1, 'b1, 'c1, 'd1, 'e1, 'f1,
     'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel =
  | Char_ty :                                       (* %c  *)
      ('a1, 'b1, 'c1, 'd1, 'e1, 'f1,
       'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel ->
      (char -> 'a1, 'b1, 'c1, 'd1, 'e1, 'f1,
       char -> 'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel
  | String_ty :                                     (* %s  *)
      ('a1, 'b1, 'c1, 'd1, 'e1, 'f1,
       'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel ->
      (string -> 'a1, 'b1, 'c1, 'd1, 'e1, 'f1,
       string -> 'a2, 'b2, 'c2, 'd2, 'e2, 'f2) fmtty_rel
```

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

`Format`'ing text

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

```ocaml
module E = struct
  type t =
    | Int of int
    | Add of t * t

  let rec pp_expr ppf = function
    | Int n -> Format.fprintf ppf "%02d" n
    | Add (e1, e2) ->
       Format.fprintf ppf "%a +@ %a" pp_expr e1 pp_expr e2
end
```

```ocaml
let () =
  let open E in
  List.fold_left (fun e n -> Add (Int n, e)) (Int 0) (1 |-> 20)
  |>  Format.printf "@[<hov>%a@]@." pp_expr
```

```
20 + 19 + 18 + 17 + 16 + 15 + 14 + 13 + 12 + 11 + 10 + 09 + 08 + 07 + 06 +
05 + 04 + 03 + 02 + 01 + 00
```

For more: [2]

Polymorphic variants i

```ocaml
type const = [ `True | `False ]

(* See e.g., https://en.wikipedia.org/wiki/NAND_logic *)
let rec nandify = function
  | #const as b -> b
  | `Bnot b ->
     let b' = nandify b in `Bnand (b', b')
  | `Band (b1, b2) ->
     let b1 = nandify b1 and b2 = nandify b2 in
     `Bnand (`Bnand (b1, b2), `Bnand (b1, b2))
  | `Bnand (b1, b2) ->
     `Bnand(nandify b1, nandify b2)
  | `Bor (b1, b2) ->
     let b1 = nandify b1 and b2 = nandify b2 in
     `Bnand (`Bnand (b1, b1), `Bnand (b2, b2))
```

Polymorphic variants ii

```
type const = [ `False | `True ]
val nandify :
  ([< `Band of 'a * 'a
    | `Bnand of 'a * 'a
    | `Bnot of 'a
    | `Bor of 'a * 'a
    | `False
    | `True ]
   as 'a) ->
  ([> `Bnand 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

Give it a try

`OCaml`

A pragmatic functional language

Try it at `https://ocaml.org/`

Final remark

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.