Some random tricks in OCaml (mostly about modules)
Because not everyone reads the whole manual (neither do I)

Using a language for several years allows you to discover some subtleties

- Some may seem naive (it is more about tricks)
- It is not about advanced stuff in the type-system
- You know that my english sucks (sorry), and we have time so don't worry about interrupt me!

Plan

- Some recap about modules (as a **language feature** and not as a compilation feature)
- A small use-case of GADTs usage (which is not *really* about type equalities)
- A (very) small consideration about OOP on front of GADTs and existentials

Modules

*OCaml, **O**bjects **C**an be **A**voided using the **M**odule **L**anguage*

A slightly false sentence

Modules (as a language): core concepts

Module language is a small functional programming language (`Fw`) in OCaml.

- **Encapsulation**: separate **implementation** (`struct`) and **description** (`sig`) (allowing **abstraction** and partial abstraction (`private`))
- **Generalization**: you can define **transluent signatures**
- **Poor's man Namespacing**
- **Typing features**: enables **Higher Kinded Polymorphism**
- **Functional**, there are function **from module to module** (that can be *applicative* or *generative*)
- **First class**: modules can interact with the value level (with some restriction on kind arities)
- **Recursive**, useful but... *urgh*

Encapsulation

```
type 'a t

val empty : 'a t
val push : 'a -> 'a t -> 'a t
val pop : 'a t -> 'a option * 'a t
```

```
type 'a t = 'a list

let empty = []
let push x s = x :: s

let pop = function
  | [] -> None, empty
  | x :: xs -> Some x, xs

let some_internal_function = ...
```

Transluent Signatures

*A **signature that is not already attached to a concrete module***

```
module type STACK = sig
  type 'a t

  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a option * 'a t
end
```

```ocaml
module Stack_list : STACK = struct
  type 'a t = 'a list

  let empty = []
  let push x s = x :: s

  let pop = function
   | [] -> None, empty
   | x :: xs -> Some x, xs

   let some_internal_function = ...
end
```

As our signature is defined as being attachable to several concrete modules, the type is ... *defacto* unknown. So abstract by default. Sometimes we'd like to provide a concrete representation of our types, even though we use a **module type**.

```
module Stack_list : STACK with type 'a t = 'a list = struct
  type 'a t = 'a list
  ...
end
```

Now, 'a Stack_list.t' is no longer an abstract type.

Modules are not modules types and vice versa

Module can be inserted into another module

```
module My_list = struct
  include Stdlib.List
end
```

Module type can be inserted into another module type or into a signature

```
module My_stack : sig
  include STACK with type 'a t = 'a list
end = struct ... end

module type  MY_STACK = sig
  include STACK
end
```

Module type can be inserted into another module type or into a signature

```
module type MY_CUSTOM_LIST = sig
  include Stdlib.List
    (* won't work since `List` is not a module type *)
end
```

Convert concrete module to module type

```
module type MY_CUSTOM_LIST = sig
  include module type of Stdlib.List
end
```

Directly promoting module type without value to module (without value)

let's imagine that we want to convert this `S` into a module `A` (concrete)

```
module type S = sig
  type t
  type e
  type f
end
```

```
module A : S
  with type i = int
  and type s = string
  and type f = float =
struct
  type i = int
  type s = string
  type f = float
end
```

Damn. . . so verbose

```
module type A = S
   with type i = int
   and type s = string
   and type f = float
module rec A : A = A (* recursive trick *)
```

It works (well) only because there is no value into S.

Module as Namespaces

The obvious approach to using modules (before abstraction and generalisation) is to use them as **namespaces**. Since modules are enclosed in a **true language** you can write some common patterns in namespace importation.

Renaming

```
import * from List
```

Just uses open (List function won't be re-exported)

```
open List
```

Renaming

```
import Foo as Bar
```

Just uses modules aliases

```
module Foo = Bar
```

Selective importation

```
import {map, iter} from List
```

Use arbitrary modules expression in open statement

```
open (
  List :
    sig
      val map : ('a -> 'b) -> 'a list -> 'b list
      val iter : ('a -> unit) -> 'a list -> unit
    end)
```

But it can be ... anoying because you need to know/and repeat the types that you need

Use arbitrary modules expression in open statement 2

```
open struct
  let map, iter = List.(map, iter)
end
```

Selective importation with renaming

```
import {map, iter as for_each} from List
```

Use arbitrary modules expression in open statement

```
open struct
  let map, for_each = List.(map, iter)
end
```

Using arbitrary modules expression you can even abstract without mli

```
open struct
  (* My private API *)
  let make a b c = ...
end

(* my public api *)
let make a b c =
 make
   <$> validate_a a
   <*> validate_b b
   <*> validate_c c
```

Partial abstraction (private)

```
type t = private int

val make :
    int
    -> (t, [ `Invalid_age ]) Result.t
```

```
type t = int

let make x =
    if x < 0 then
        Error `Invalid_age
    else
        Ok x
```

- prohibits construction "outside the module"
- leak the representation (allowing pattern matching, for example)

An other example from lambdaLille/History

*Describes a **Talk** and everything should be validated*

Type (in mli it is private)

```
type talk = private
  { title : string
  ; speakers : Speaker.t list
  ; abstract : string option
  ; tags : string list
  ; lang : [ `French | `English ]
  ; video_link : string option
  ; support_link : string option
  ; other_links : Link.t list
  }
val from_string : (module Yocaml.Metadata.VALIDABLE)
   -> string -> (talk, Error.t) Result.t
```

In `ml`

```
type talk = ...

let make title speakers abstract tags
    lang video_link support_link other_links =
  {title; speakers; abstract; tags; lang; video_link;
    support_link; other_links}
```

```ocaml
let from_string
     (module Validable : Yocaml.Metadata.VALIDABLE) = function
   let* value = Validable.from_string str in
    object_and
     (fun obj ->
       make
        <$> required_assoc string "title" obj
        <*> required_assoc (list_of string) "speakers" obj
        <*> optional_assoc string "abstract" obj
        <*> optional_assoc_or ~default:[] (list_of string) "tags" obj
        <*> required_assoc (lang (module Validable)) "lang" obj
        (* to be continued, but slides are to small... *)
    value
```

Chosing between Abstract and Private types

Abstract types

- When you want a structure to follow a general interface
- When you know that your implementation **will change**
- Abstract types fit well with the description of **data structures**

Private types

- When the structure of the type matters
- But to restrict its creation (like in `Age.t` or `Talk.talk`)
- private types fit well with the description of **structured data attached to constraints**

Functors : function from module to module

Don't be confused with **functors** from Haskell.

```
module type MAPPABLE = sig
  type 'a t (* Notice that this is Higher Kinded Polymorphism *)
  val map : ('a -> 'b) -> 'a t -> 'b t
end

module Iterable (M: MAPPABLE) : sig
 type 'a t = 'a M.t
 val iter : ('a -> unit) -> 'a t -> unit
end = struct
 type 'a t = 'a M.t
 let iter f x =  ignore @@ M.map f x
end
```

Modules are **structurally subtyped**

```
module List_iterable = Iterable (List)
```

It works because `List` has a `'a t` and a `map` function that fit with our contract.

A trick if you hate structural subtyping

If you **hate** structural subtyping (and you prefer **nominal subtyping**):

- Nominal subtyping is a specialization of structural subtyping
- So, **having structural subtyping implies the ability to encode nominal subtyping**

```
module type MAPPABLE = sig
  type 'a t
  val map : ('a -> 'b) -> 'a t -> 'b t
end

module type NOMINAL_MAPPABLE = sig
  include MAPPABLE
  val is_mappable : unit
end
```

```
module Extend (M: MAPPABLE) :
  NOMINAL_MAPPABLE with type 'a t = 'a M.t =
struct
  include M
  let is_mappable = ()
end

( ... )

module List_iterable = Iterable (Extend (List))
```

But, *seriously*, Why would we want to do that. . .

Some syntactic analogy

Value Level

```
let f x y = ...
```

Module Level

```
module F (X: SIG_X) (Y: SIG_Y) = ...
```

Value Level

```
let g = f x
```

Module Level

```
module G = F (X)
```

Value Level

```
let f = fun x y -> ...
let g = function x -> function y -> ...
```

Module Level

```
module F = functor (X: SIG_X) (Y: SIG_Y) -> ...
module G = functor (X: SIG_X) -> functor (Y: SIG_Y) -> ...
```

Functor can be **Applicative** or **Generative**

Applicative

```
module T = struct type t = int end
module F (T : sig type t end) = struct
  type t = F of T.t
end

module A = F (T)
module B = F (T)

let x : A.t = B.F 1 (* Works *)
```

Generative

```
module T = struct type t = int end
module G (T : sig type t end) () = struct
  type t = F of T.t
end

module A = G (T) ()
module B = G (T) ()

let x : A.t = B.F 1 (* Do not Works *)
```

Generative functors generate **fresh types** at **each application**.

Chosing between Applicative and Generative

In general, Applicative (the default behaviour) is **the right choice**. But sometime, Generatives can be useful.

- When you functor is **impure**
- It fit very well with **first-class-modules** for generating fresh types (because you can unpack (in an existential sense) first-class-modules in the body of a generative functor.)
- Allows some specifics encoding like **singleton-ish** types

First classes modules

Building modules inside a value using regular values

Using first classes modules you can produce modules using user-defined values.

```
module type CONFIG = sig
  val server : string
  val port : int
end

let run (module C : CONFIG) = ... some complex code

let mk_config ~server ~port =
  let module C = struct
    let server = server
    let port = port
  end in run (module C)
```

First classes modules

Parametric function

When you don't deal with parametrized type it works well.

```
module type TO_STRINGABLE = sig
  type t
  val to_string : t -> string
end

let print
    (type s)
    (module S : TO_STRINGABLE with type t = s)
    (s : s) =  print_endline @@ S.to_string s
```

When use First Class modules

First class modules are **existentials** so they make sense when you need existentials

- If your values (inside the module) depends on user-given values
- when you need to deal with fresh types using generativity
- for API purpose (when you do not deal with kind upper zero)

If you absolutely want FCM, you need to provide Functor for monomorphization and it can be cumbersome

Static finite state machines

Sometimes, the **flow** of your application can be modeled as a **finite state machine**.

- A sequence of operation, loop and conditionals is an implicit state machine
- Sometime state machine are defined inside the program flow.

The second point is about dynamic states machines (and is well solved by machine over some complex categorical stuff, ie: **Mealy/Moore Machine** over Profunctors and Comonad).

The first one is solved by being explicit about the state machine inside the structure of your program.

Let's imagine a very simple state machine

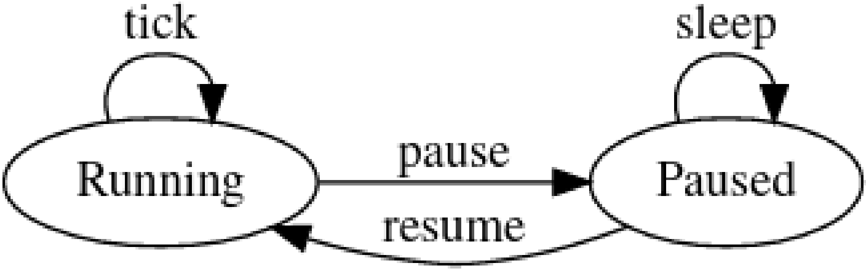

Figure 1: lol, sorry for the definition

You can modeling transition using GADT

GADTs

It just sum types that allows constructor to be not surjective and that introduces local types equalities (that made existentials easy to represents).
In other words, GADts allows (among others) to define sum constructors that are indexed by types witnesses.

```
type _ t =
  | Foo : float t
  | Bar : int t
```

- `Foo` has type `float t`
- `Bar` has type `int t`

When we read paper or document about GADTs, it is always about complicated (but interesting stuff). Here, is a very simple example that show how to produces type safe finite state machine using GADTs.

```
let f = function
  | Foo -> "hey"

let g = function
  | Bar -> "hoy"
```

Thanks **local type equalities**, we can have exhaustive pattern matching that don’ t match very constructor.

For expressing our transitions we need rows

Just indexing types with regular types as witnesses leads to a lack of flexibility. So we should uses **rows**. We are lucky, OCaml allows two define rows using two ways :

- Rows on products are objects (so yes, Objects are pretty good as phantom type parameter)
- Rows on sums are polymorphic variants

For this example, we will use polymorphic variants.

```ocaml
type time = int

type _ state =
  | Running : time -> [ `Running ] state
  | Paused : time -> [ `Paused ] state

let start () = Running 0
let resume (Paused x) = Running x
let pause (Running x) = Paused x
let tick (Running x) = Running (x + 1)
```

```
let a =
    start ()
    |> tick
    |> pause
    |> resume
    |> tick

# [ `Running ] state = Running 2
```

```
let b =
  start ()
  |> tick
  |> pause
  |> sleep 10
  |> resume
  |> tick

# [ `Running ] state = Running 12
```

```
let c =
  start ()
  |> sleep 10

# BOUM. It does not work. muehehe
```

So since we know GADTs and Modules, we do not need objects ! Yes, with modules you can have

- abstraction
- encapsulation
- verbose existentials

and with GADTs you can have

- existentials that introduces type equalities. So !

But what are objects

> *In the OOP world, having a good definition of Object is . . . complicated. But not only in a technical jargon, in french the most generic definition is : "**Something to look at**" (in Roland Barthes "L'aventure sémiologique").*

> *Fortunately, Alan Kay, one of the father of the OOP says that an object is . . . "**that can can receive messages**"*

So giving a good definition of Objects, in the sense of OOP, that works everywhere seems impossible so let's gives a . . . OCaml definition.

Objects are **products** that introduces **open recursion** and lexically scoped **self value**, that allows, by **late-binding** *a posteriori* specialization and that **universally quantify over type parameters** (generics) and **existentially over the polymorphic variables of methods**.

So . . . when use objects ?

- Object types as a phantom type parameters
- and . . . if you write code like this:

```
type t =  T : ((module S with type t = 'a) * 'a) -> t
```

You are smart, but **you definitively needs objects**

FIN

thanks