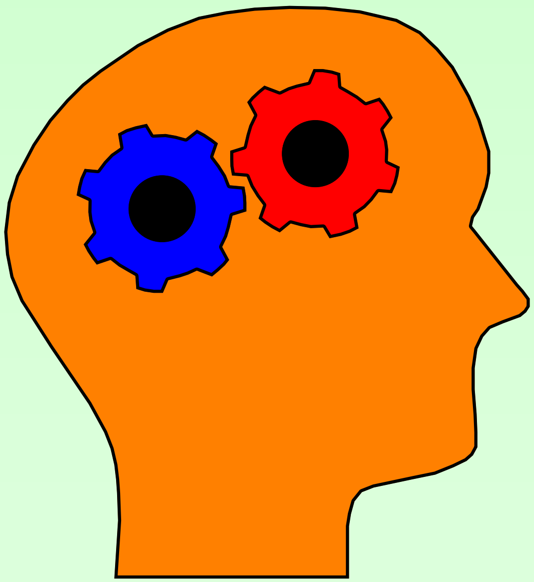
CS2104: Programming Languages Concepts

Lecture 11 : *OOP and Modules in OCaml*



“OOP and Modules”

Lecturer : <u>Chin</u> Wei Ngan

Email : chinwn@comp.nus.edu.sg

Office : COM2 4-32

Topics

- Classes & Objects
- Structural Types and Row Polymorphism
- Modules : Structure and Signature
- ADT
- Functors

Classes and Objects

OCaml Class

- Each class comprise of fields and methods.

```
class counter =
 object
  val mutable x = 0
  method inc = x <- x + 1
  method get = x
  method set y = x <- y
 end;;
```

field

methods

- Class is a factory of objects:

```
let p = new counter;;
let q = new counter;;
p # inc;;
q # set 5;;
```

Object

- We can create a single object (without class), as follows.

```
let p =
 object
  val mutable x = 0
  method inc = x <- x + 1
  method get = x
  method set y = x <- y
end;;
```

- This is similar to a singleton class in Scala, but is actually closer to an anonymous class.

Class Parameters

- Class may have parameters for its class constructors.

```
class counter init =
 object
  val mutable x = init
  method inc = x <- x + 1
  method get = x
  method set y = x <- y
 end;;
```

- Type of class constructor would be a function.

```
class counter : int -> object … end
```

- Note that class name is abbreviation of object type itself.

Reference to Self/This

- You can explicitly name the current object.

```
class count_step init step =
 object (s)
  val mutable x = 0
  method inc = x <- x + step
  method get = x
  method print = string_of_int (s # get)
 end;;
```

name for
this object

- This a named version of the current "this" object in Java/Scala.

Class Inheritance

- We can use class inheritance to obtain *fields* and *methods* of prior (super) class, and to support overriding.

```
class count_step init step =
 object (s)
  inherit counter init
  method inc = x <- x + step
  method print = string_of_int (s # get)
 end;;
```

Class Polymorphism

- We can support polymorphic classes using type variables.

- A simple generic buffer.

```
class ['a]  buffer init =
 object
   val mutable value :'a = init
   method get = value
   method set n = value <- n
end;;
```

Class Signature

- Class type is based on structure of the set of *visible* methods.

- Type signature of generic buffer (*without* fields as they are always hidden).

```
class type ['a]  buffer_type =
 object
   method get : `a
   method set : `a -> unit
 end;;
```

Structural Type Equivalence

- Another equivalent generic buffer class is:

```
class ['a]  buffer2 init =
 object
   val mutable value :'a = init
   val mutable is_empty = false
   method get =
     if is_empty then failwith "empty buffer"
     else (is_empty <- true; value)
   method set n = (value <- n; is_empty <- false)
   method private reuse = is_empty <- false
 end;;
```

- This has the same set of visible methods (with the same type) as the earlier buffer_type.

Structural Typing

- Two types are the same if they are *structurally equivalent* to each other on the *visible* methods.

```
let foo (v:'a buffer_type) = v # get;;
(* foo : 'a1 buffer_type -> 'a1 *)

let v = new buffer 5;;

let w = new buffer2 5;;
```

- Though v and w are created by different classes, they have the *same* type signature.

Subtyping via Row Polymorphism

- Structural subtyping is supported via "*row polymorphism*".

```
let foo2 (v) = v # get;;
 (* foo2 :  < get : 'b; .. > -> 'b *)

let v = new buffer 5;;
let w = new buffer2 5;;
foo2 v;;
foo2 z;;
```

- The row polymorphic type `<get : 'b; .. >` will unify with any type with a get method in its class type.

- The matched type need not be structurally equivalent.

Subtyping via Coercion

- It is possible to coerce type via up-cast operation.

```
let foo3 (v) = (v:>'b buffer) # get;;
 (* foo3 :  `b #buffer -> 'b *)

let v = new buffer 5;;
let w = new buffer2 5;;
foo3 v;;
foo3 z;;
```

- The will unify with any type that contains at least the visible methods from the buffer class.

- The matched type is a structural subtype.

Object Cloning

- An object can be cloned using Oo.copy which will make an new instance of its field variables. This is a *shallow* copy.

```
let foo2 (v) = v # get;;

let w = new buffer2 5;;
let z = Oo.copy w;;
(* Oo.copy :  (< .. >  as 'a) -> 'a *)

foo2 v;;
foo2 z;;
```

- Deeper sharing are still present in the copied object.

- Note that `Oo.copy` works with any object type `< .. >`.

Functional Objects

- If we *disallow* mutability of fields, we can get *functional* objects.

```
class func_point y =
object
  val x = y
  method move (d:int) : func_point =
    new func_point (x+d)
end;;
```

- Note that this is also a *recursive* class type. What is its class type signature?

Mutually-Recursive Classes

- Mutual-recursive types can also be defined.

```
class window =
  object
  val mutable top_widget
      = (None : widget option)
  method top_widget = top_widget
  end
and widget (w : window) =
  object val window = w
   method window = window
  end;;
```

- The class signature would be mutual-recursive too.

Virtual Methods and Classes

- We can define a class with some *undefined* methods.

```
class virtual ['a]  buffer_eq init =
  object (this)
    val mutable value :'a = init
    method get = value
    method virtual eq : 'a buffer_eq -> bool
    method neq b = not(this # eq b)
  end;;
```

- Objects of virtual (undefined) classes cannot be instantiated.

- This is the same as abstract classes and abstract methods in Scala.

Implementing Virtual Methods

- Concrete classes should give definitions for its virtual methods.

```
class ['a]  buffer init =
 object (this)
  inherit ['a] buffer_eq init
  method eq that
      = this # get  = that # get
  method set n = value <- n
end;;
```

- Class instantiation now possible with concrete classes.

```
let b = new buffer 5
let c = new buffer_eq 5 (* invalid *)
```

Modules
& Functors

Modules

- OCaml has an *advanced* module system.
- Modules (like Scala/Java package) used to structure large programs.
- Modules allow us to conserve name space.
- Modules allow us to support ADT.

Module

- Modules can be used to group *types*, *values*, *functions*, *exceptions* and other *modules* together.

```
module Buffer =
struct
  type 'a t = ('a option) ref
  let emp () : 'a t = ref None
  let get buf = match !buf with
      | None -> failwith "Buffer_Err"
      | Some r -> (buf := None; r)
  let put buf x = buf := Some x
end
```

- A *structure* is an implementation for module.

Qualified Names

- We use qualified names to access values.

```
type ‘a tt = ` a Buffer.t
let g = Buffer.get
```

- However, we may open module to have its entities visible locally, without any qualifiers.

```
open Buffer
type ‘a tt2 = ` a t
let g = get
```

Module Signature

- Each module has a type signature that can also be explicitly declared. For example:

```
module type BUFFER =
sig
  type 'a t = ('a option) ref
  exception Buffer_Err
  val emp : unit -> 'a t
  val get : 'a t -> 'a
  val put : 'a t -> 'a -> unit
end
```

- Convention to write module type entirely in upper-case.

- A type signature is an interface for a module.

Module ADT

- We can hide implementation details by using a more *abstract* type signature.

```
module type BUFFER_ABS =
sig
  type 'a t
  val emp : unit -> 'a t
  val get : 'a t -> 'a
  val put : 'a t -> 'a -> unit
end
```

- Type `‘a t` is now made abstract with two implementation details hidden
  (i) option ref (ii) Buffer_Err exception

Module ADT

- Type of ADT is actually an existential type.

```
module type BUFFER_ABS =
sig
  type 'a t
  val emp : unit -> 'a t
  val get : 'a t -> 'a
  val put : 'a t -> 'a -> unit
end
```

Similar to existential type:

```
∃t. { emp : unit -> 'a t;
      get : 'a t -> 'a;
      put : `a t -> `a -> unit }
```

Abstract Data Type Implementation

- We can provide implementation for abstract modules.

```
module BufferADT = Buffer : BUFFER_ABS
```

- Without information hiding, we can access more things.

```
let b1 = Buffer.Buffer_Err
let b2 = !(Buffer.emp ()) == None
```

- Using abstract module **BufferADT**, we disallow implementation details to be exposed

```
let b1 = BufferADT.Buffer_Err (* invalid *)
let b2 = !(BufferADT.emp ()) == None (* invalid *)
```

Abstract Data Type Implementation

- Module implementation can be directly *associated* with ADT type.

```
module Buffer2 : BUFFER_ABS  =
struct
  type 'a t = ('a option) ref
  exception Buffer_Err
  let emp () : 'a t = ref None
  let get buf = match !buf with
      | None -> raise Buffer_Err
      | Some r -> (buf := None; r)
  let put buf x = buf := Some x
end
```

Alternative ADT Implementation

- We can choose other kinds of implementation, such as unbounded *mutable list* for our buffers.

```
module BufferL : BUFFER_ABS =
struct
  type 'a t = ('a list) ref
  let emp () = ref []
  let get buf = match !buf with
      | [] -> failwith "empty"
      | r::xs -> (buf := xs; r)
  let put buf x = buf := (!buf@[x])
end
```

Functor

- Functors are functions from structures to structures.
- Functor for priority buffer.

```
module Buffer_P =
 functor (Elt: PRIORITY_TYPE) ->
   struct
   type element = Elt.t
   type t = (element list) ref
     :
   let put buf x = buf := ins !buf x (Elt.get_p x)
end;;
```

Module Type for Priority

- Module Type.

```
module type PRIORITY_TYPE =
sig
  type t
  val get_p : t -> int
end;;
```

- Example 1 :

```
module Int : PRIORITY_TYPE =
   struct
   type t = int
   let get_p x = x
end;;
```

- Example 2 :

```
module Int_P : PRIORITY_TYPE =
   struct
   type elm
   type t = elm * int
   let get_p (_,x) = x
end;;
```

Usage of Functors

- Specializing two different modules with Functors

```
module PQ1 = Buffer_P(Int)

module PQ2 = Buffer_P(Int_P)
```

- Notice that Module generated from Functor application.
- Supports code reuse via modules.