EPITA - Practical Programming
05 - The O in OCaml

Marwan Burelle

marwan.burelle@lse.epita.fr
Outline

1 Object Oriented Programming
   - Objects ?
   - Inheritance ?

2 OCaml’s OO
   - OCaml’s Classes
   - Inheritance
   - Inheritance And Subtyping
   - Late Binding
   - Multi-inheritance
   - Virtual Classes
   - Type Classes, Derived Types And Typing Issues

3 Advanced OOP Features In OCaml
   - Polymorphic Classes
   - Polymorphic Object
   - Modules And Objects
Object Oriented Programming
- **OOP**: Object Oriented Programming
- Complementary concept about code and data organization
- Object extensions in the language are not required but provide comfort.
- Supposed to provide:
  - Better organization
  - Better code reuse
  - Simpler and safer development
- Languages with OOP features: Smalltalk, C++, Java, C#, OCaml, D, Go, ...
1 Object Oriented Programming
   - Objects?
   - Inheritance?
From Abstract Types To Objects

- We start with Dijkstra’s abstract types: data are described through their operations.
- Objects is the next step in abstraction: we embed operations inside the data and (often) we expose data only through the mean of the operations. This is the encapsulation principle.
Objects And Classes

- **An object** is a block of structured data together with the operations to manipulate these data.

- **A class** is a description of a family of objects: it describes the layout of the content (data) and the code of the associated operations (member functions and/or methods.)

- **An instance** is an object built from a class description.

- **A method** (or a message) is an operation provided by an object, the *caller* only provides the name of the operation and its parameters, while the target object (receiver) selects the corresponding code to be executed.

- **A member function** is a poor form of method (used to avoid the extra cost of the late binding.)
1 Object Oriented Programming
- Objects?
- Inheritance?
• When modeling complex systems, it often appears that entity can be described as generalization or specialization of other entities (such as cars are specialization of vehicles, or animals are generalization of cats.)

• Many OO languages provides a way to use this relation, this is inheritance

• Inheritance provides nice modelizations and emphases code reuse.

• Smart usage of inheritance (and associated notions) often permits fast and reliable software design and development.
The O in OCaml

- OCaml provides an Object Oriented layer
- It's a classical class based OO extension with: multiple-inheritance, dynamic method selection (partial late binding), method overriding, class polymorphism and instance polymorphism, subtyping . . .
- While not the ultimate OO language, all these extensions make OCaml a comfortable one.
Overview

2 OCaml’s OO
  ■ OCaml’s Classes
  ■ Inheritance
  ■ Inheritance And Subtyping
  ■ Late Binding
  ■ Multi-inheritance
  ■ Virtual Classes
  ■ Type Classes, Derived Types And Typing Issues
A First Class

(* points.ml *)

class point =
object
  val mutable x = 0
  val mutable y = 0
  method getX = x
  method getY = y
  method setX x' = x <- x'
  method setY y' = y <- y'
end
A First Instance

(* points.ml *)

let p = new point
let main () =
  begin
    Printf.printf "(%d,%d)\n" p#getX p#getY;
p#setX 1;
    Printf.printf "(%d,%d)\n" p#getX p#getY;
    exit 0;
  end
let _ = main ()

> ./points.native
(0,0)
(1,0)
class point x' y' =
object
    val mutable x = x'
    val mutable y = y'
    method getX = x
    method getY = y
    method setX x' = x <- x'
    method setY y' = y <- y'
end
File "points.ml", line 1, characters 6-162:
Error: Some type variables are unbound in this type:
    class point :
        'a ->
        'b ->
    object
        val mutable x : 'a
        val mutable y : 'b
        method getX : 'a
        method getY : 'b
        method setX : 'a -> unit
        method setY : 'b -> unit
    end
The method getX has type 'a where 'a is unbound
Class With Parameters

First attempt: add operation on attribute to fix type

```ocaml
class point x' y' =
object
  (* dirty trick *)
  val mutable x = x' + 0
  val mutable y = y' + 0
  method getX = x
  method getY = y
  method setX x' = x <- x'
  method setY y' = y <- y'
end
```
Better solution: add type on attribute

```ocaml
class point x' y' =
  object
    val mutable x : int = x'
    val mutable y : int = y'
    method getX = x
    method getY = y
    method setX x' = x <- x'
    method setY y' = y <- y'
  end
```

Instance With Parameters

(* points.ml *)

let p = new point 10 10
let main () =
    begin
        Printf.printf "(%d,%d)\n" p#getX p#getY;
        p#setX 1;
        Printf.printf "(%d,%d)\n" p#getX p#getY;
        exit 0;
    end
let _ = main ()

> ./points.native
(10,10)
(1,10)
Auto-reference

class point x' y' =
object (self)
    val mutable x:int = x'
    val mutable y:int = y'
method getX = x
method getY = y
method setX x' = x <- x'
method setY y' = y <- y'
method dist2 = (x*x + y*y)
method dist = sqrt (float self#dist2)
end
Auto-reference

Auto-reference is free!

```ocaml
class a =
object (this)
    method m = 21
    method answer = 2 * this#m
end

class b =
object (self)
    method m = 21
    method answer = 2 * self#m
end

class c =
object (me_myself_and_I)
    method m = 21
    method answer = 2 * me_myself_and_I#m
end
```
More Complete Example

A Simple Evaluation State

class state =
object (self)
  val store = Hashtbl.create 101
  val mutable size = 0
  method mem (key:int) =
      Hashtbl.mem store key
  method add key (value:int) =
      begin
          Hashtbl.replace store key value;
          if not (self#mem key) then
              size <- size + 1
      end
  method find key =
      Hashtbl.find store key
end
Visibility

• In OCaml there’s only two possible visibilities: private or public.
• Attributes are always private.
• Methods are public by default but can be declared as private explicitely.
• If you need finer visibility, you can play with modules interfaces.

```ocaml
class a =
  object
    (* private attribute *)
    val mutable x = 0
    (* public method *)
    method get = x
    (* private method *)
    method private double = x <- x * x
  end
```
2 OCaml’s OO

- OCaml’s Classes
- Inheritance
- Inheritance And Subtyping
- Late Binding
- Multi-inheritance
- Virtual Classes
- Type Classes, Derived Types And Typing Issues
Basic Inheritance

--- Colored Points ---

class point x' y' =
object (self)
  val mutable x : int = x'
  val mutable y : int = y'
method getX = x
method getY = y
method setX x' = x <- x'
method setY y' = y <- y'
end

class colored_point x' y' col =
object (self)
  inherit point x' y'
  val mutable color : string = col
  method getColor = color
  method setColor col = color <- col
end
Basic Inheritance

let main () =
  begin
    let p = new colored_point 0 0 "black" in
    Printf.printf "(%s,%d,%d)\n"
    p#getColor p#getX p#getY;
    exit 0;
  end
let _ = main ()

> ./points.native
(black,0,0)
Overriding

Overriding a method

```ocaml
class point x' y' =
object (self)
  val mutable x : int = x'
  val mutable y : int = y'
  method to_string =
      Printf.sprintf "(%d,%d)" x y
end

class colored_point x' y' col =
object (self)
  inherit point x' y'
  val mutable color : string = col
  method to_string =
      Printf.sprintf "(%s,%d,%d)" color x y
end
```
Overriding

let main () =
  begin
    let p = new point 0 0 in
    let p' = new colored_point 0 0 "black" in
    Printf.printf "p: %s\n" p#to_string;
    Printf.printf "p': %s\n" p'#to_string;
    exit 0;
  end
let _ = main ()

> ./points.native
p: (0,0)
p': (black,0,0)
Overview

2 OCaml’s OO

- OCaml’s Classes
- Inheritance
- Inheritance And Subtyping
- Late Binding
- Multi-inheritance
- Virtual Classes
- Type Classes, Derived Types And Typing Issues
Subtyping?

- Subtyping describes the idea that values of a given type can be used where a less specific type is expected.
- Let $t$ and $t'$ be two type, $t'$ is a subtype of $t$, noted $t' \leq t$, if $\forall x \; x : t' \Rightarrow x : t$. Any value of type $t'$ can be seen of type $t$.
- Theoretically, when a class $B$ inherit from a class $A$, we may have $B \leq A$.
- Beware, inheritance doesn’t always imply subtyping!
- In OCaml, subtyping is only available for classes and polymorphic variant.
- Being a well found strongly typed language, OCaml enforces strictly the difference between subtyping and inheritance (not the case in many languages.)
When typing OOP languages, there’s two main strategies: structural typing and class name typing.

In class name typing, only class names are used in subtyping and thus an object belongs to a type only if it was declared using that type (or a subtype.)

In structural typing, the (visible) content of an object defines its type and thus an object belongs to any types providing the same content.

OCaml uses structural typing.
OCaml type inference combined with structural typing offers simple and powerful flexibility: objects as function’s parameters are only required to provide the methods passed to the parameters.

```
let f o = 1 + o#m 1;;
# val f : < m : int -> int; .. > -> int = <fun>
```

The function \( f \) is accepting any object providing a method \( m \) that take an \( \text{int} \) and return an \( \text{int} \).
Example

```ocaml
let f o = 1 + o#m 1
class a =
  object
    method m x = 2 * x
    method m' x = 3 * x
  end

class b =
  object
    method m x = 4 * x
  end

let main () =
  begin
    let (a, b) = (new a, new b) in
    Printf.printf "%d
" (f a + f b);
    exit 0;
  end

let _ = main ()
```
2 OCaml’s OO

- OCaml’s Classes
- Inheritance
- Inheritance And Subtyping
- Late Binding
- Multi-inheritance
- Virtual Classes
- Type Classes, Derived Types And Typing Issues
Late Binding?

- In OOP method resolution can be done at compile time or at run time.
- Normally, choosing the right version of the method is the responsibility of the object: this is *simple dynamic dispatch* a form of late binding.
- If the language offers overloading, there’s the same issue about the overloaded version (anyway, OCaml doesn’t provides overloading.)
- The main impact is that you know that the method called will always be the version defined in the object.
Example

```ocaml
class animal m s =
obj
  val member = m
  val scream = s
  method describe =
    Printf.printf "This animal has %d legs and he %s\n"
    member scream
end

class dog b =
obj
  inherit animal 4 "bark"
  val breed = b
  method describe =
    Printf.printf "This dog is a %s.\n" breed
end
```
Example

```
let main () =
let (a:animal) = new dog "Beagle" in
let (b:animal) = new animal 4 "mew" in
begin
  a#describe;
  b#describe;
  exit 0;
end
let _ = main ()

> ./animal.native
This dog is a Beagle.
This animal has 4 legs and he mew
```
Overview

2 OCaml’s OO
- OCaml’s Classes
- Inheritance
- Inheritance And Subtyping
- Late Binding
- Multi-inheritance
- Virtual Classes
- Type Classes, Derived Types And Typing Issues
Inheriting From Different Classes

```ocaml
open Printf
class mamal furcolor =
object
    method describe =
        printf "A mamal with %s fur.\n" furcolor
end
class flyer =
object
    val mutable x = 0
    val mutable y = 0
    method flyto x' y' =
        printf "Fly from (%d,%d) to (%d,%d)\n" x y x' y'; x <- x'; y <- y'
    method describe =
        printf "Currently at (%d,%d)\n" x y
end
```
Inheriting From Different Classes

class bat =
object
  inherit mamal "black"
inherit flyer
end

let main () =
  let b = new bat in
  b#describe;
let _ = main ()
Order Matters

--- bat.ml ---

class bat =
object
  inherit flyer
  inherit mamal "black"
end

let main () =
  let b = new bat in
  b#describe;
let _ = main ()

> ./bat.native
A mamal with black fur.
Referencing Parents

```ocaml
class bat =
object
  inherit flyer as f
  inherit mamal "black" as m
  method describe =
    m#describe; f#describe
end

let main () =
  let b = new bat in
  b#describe;
  let _ = main ()

> ./bat.native
A mamal with black fur.
Currently at (0,0)
```
2 OCaml’s OO

- OCaml’s Classes
- Inheritance
- Inheritance And Subtyping
- Late Binding
- Multi-inheritance
- Virtual Classes
- Type Classes, Derived Types And Typing Issues
Incomplete Classes?

- Sometimes, it can be useful to provide an incomplete description of a class.
- That’s what are called virtual (OCaml) or abstract (Java/C#) classes.
- The idea is to provide code and constraints: some operations are not defined (no code) but used in the code provided.
- In OCaml incomplete classes are tagged as virtual.
- Virtual classes contains virtual methods: methods declared but not defined.
- Such classes can’t be instantiated.
- When inheriting from a virtual class: you must provide code for the virtual methods or the inherited class must be virtual as well.
class virtual printable =
  object (self)
    method virtual to_string : string
    method print =
      Printf.printf "obj: %s\n" self#to_string
  end

class printable_point x’ y’ =
  object (self)
    inherit printable
    val mutable x = x’
    val mutable x = y’
    method to_string =
      Printf.sprintf "(%d,%d)" x y
  end
class virtual err_printable =
object (self) inherit printable
    method err_print =
        Printf.eprintf "<DEBUG>obj: %s\n"
        self#to_string
end

class point3D x' y' z' =
object
    val x = x' val y = y' val z = z'
    method to_string =
        Printf.sprintf "(%d,%d,%d)" x y z
end

class printable_point3D x y z =
object
    inherit point3D x y z
    inherit err_printable
end
2 OCaml’s OO

- OCaml’s Classes
- Inheritance
- Inheritance And Subtyping
- Late Binding
- Multi-inheritance
- Virtual Classes
- Type Classes, Derived Types And Typing Issues
Class Interfaces

• In most OOP languages, we use interfaces to describe outer behavior of objects.
• Structural typing and type inference remove most usage of interfaces, but it can be *comfortable* to have them.
• For that OCaml provides *class types* that describe methods like in virtual classes.
• Class types can then be used as field types or input types . . .
class type animal =
  object
    method describe : unit
  end
let dum_animal = (* dummy object *)
object
  method describe = printf "Dummy Animal\n"
end
class crowd sz =
object
  val store : animal array = Array.make sz dum_animal
  val mutable size = 0
  method describe =
    for i=0 to size - 1 do
      store.(i)#describe; Printf.printf "\n"
    done
  method add (a:animal) =
    if size < sz then
      (store.(size) <- a; size <- size + 1;)
end
class type animal =
object
  method describe : unit
end

let (b:animal) = new bat

Error: This expression has type bat but an expression was expected of type animal
The second object type has no method flyto
class type animal =
object
  method describe : unit
end

(* upcast a bat into an animal *)
let b = (new bat :> animal)
More usage

```ocaml
(* officially of type animal *)
class dog : animal =
object
  method describe =
    printf "I’m a dog.\n"
end
let main () =
  let c = new crowd 10 in
  c#add (new bat :> animal);
  c#add (new dog);
  c#add (new dog);
  c#describe
let _ = main ()
```
> ./bat.native
A mamal with black fur.
Currently at (0,0)

I’m a dog.

I’m a dog.
Advanced OOP Features In OCaml
Overview

3 Advanced OOP Features In OCaml
- Polymorphic Classes
- Polymorphic Object
- Modules And Objects
Polymorphic Content

Boxes

class [ 'a ] box x =
object
  val mutable content : 'a = x
  method get = content
  method set x = content <- x
end

let intbox = new box 42
let strbox = new box "42"

typing

val intbox : int box
val strbox : string box
More Than One Variable

```ocaml
class ['a,'b] bibox x y =
object
  val mutable a : 'a = x
  val mutable b : 'b = y
  method get_a = a
  method set_a x = a <- x
  method get_b = b
  method set_b x = b <- x
end
```
class ['a] objlist v =

object (self : 'self)
    val mutable next : 'self option = None
    val mutable value : 'a = v

method get = value
method set v = value <- v

(* using optional parameters *)
method set_next ?nxt () =
    next <- nxt
method next = next
end
**Polymorphism And Methods**

---

**Generic Stores**

```ocaml
class ['a] store sz =

object
  val tab : 'a array = Array.make sz (Obj.magic None)
  val mutable size = 0
  method add x =
    (size < sz) && (
      tab.(size) <- x;
      size <- size + 1;
      true )
  method describe =
    for i = 0 to size - 1 do
      tab.(i)#describe
    done
end
```
The constraint line indicate that a method describe is required for our polymorphic content.
In the previous example OCaml inferred a constraint on the type variable.

The constraint give a more precise form to the type variable, in our case we got: `< describe : 'b; .. >`

This constraint express the fact that stored objects must provide a method describe.

Like other in cases, the inference provides the convenient answer for most situations.

Of course, you can provide your own constraints.
Enforcing Types

---

**Generic Stores**

class ['a] store sz =

object

    constraint 'a = < describe : unit; .. >

    val tab : 'a array =
        Array.make sz (Obj.magic None)

    val mutable size = 0

    method add x =
        (size < sz) && (
            tab.(size) <- x;
            size <- size + 1;
            true )

    method describe =
        for i = 0 to size - 1 do
            tab.(i)#describe
        done

end
Dealing With Optional Parameters

```ocaml
class a =
    object
      method print ?i () =
        match i with
        | None -> Printf.printf "Nothing\n"
        | Some k -> Printf.printf "Int: %d\n" k
    end

class ['a] b =
    object
      method foo (o:'a) = o#print ()
    end

(* Testing *)
let _ = (new b)#foo (new a)
```
Dealing With Optionnal Parameters

Error: This expression has type a but an expression was expected of type
< print : unit -> 'a; .. >
Types for method print are incompatible

In the context of class b the method print is seem as only accepting a unit parameter.
Dealing With Optional Parameters

--- Optional Parameters ---

```ocaml
class a =
  object
    method print ?i () =
      match i with
      | None -> Printf.printf "Nothing\n"
      | Some k -> Printf.printf "Int: %d\n" k
  end

class ['a] b =
  object
    constraint 'a=<print: ?i:int -> unit -> 'b;..>
    method foo (o:'a) = o#print ()
  end

(* Testing *)
let _ = (new b)#foo (new a)
```
Inheriting From Polymorphic Classes

Double Linked List

class ['a] doublelist v =
object (self : 'self)
  inherit ['a] objlist v
  val mutable prev : 'self option = None
  method set_prev ?p () =
    next <- p
  method prev = prev
end
Inheriting From Polymorphic Classes

Fixing inherited types

```ocaml
class ['a] box x =
  object
    val mutable content : 'a = x
    method get = content
    method set x = content <- x
  end

class intbox x =
  object
    inherit [int] box x
  end
```
Overview

3 Advanced OOP Features In OCaml
- Polymorphic Classes
- Polymorphic Object
- Modules And Objects
• In the previous examples, the classes are polymorphic but not the object.
• Sometime we may need polymorphism in method, that’s also possible!

```ocaml
class type a =
  object
    method id : 'a -> 'a
  end

class b =
  object (self : #a)
    method id x = x
  end
```
Polymorphic Object

Debug Printer

```ocaml
open Format

class type printer = object
  method print : ('a, formatter, unit) format -> 'a
end

class debug_printer fmt = object (self : #printer)
  method print fmt_str =
    fprintf fmt (@[<b 2>DEBUG:@;"
                   ^^ fmt_str
                   ^^ "[@@."
    end
end
```
Polymorphic Object

Debug Printer

```ocaml
let debug = new debug_printer err_formatter

let main () =
  begin
    debug#print "(%d,%d)" 1 42
  end
let _ = main ()

> ./debug_printer.native
DEBUG: (1,42)
```
3 Advanced OOP Features In OCaml
- Polymorphic Classes
- Polymorphic Object
- Modules And Objects
Masking Operations

```ocaml
class account =
object (self)
  val mutable value = 0
  method unsafe_op a
    value <- value + a
  method op a =
    if value + a >= 0 then
      self#unsafe_op a
  end

class manager =
object (self)
  val account = new account
  method movement a =
    account#unsafe_op (-a);
    account#unsafe_op a
  end
```
Masking Operations

---------- account.mli ----------

class account :
  object
    method op : int -> unit
  end

class manager :

  object
    method movement : int -> unit
  end