# Dynamic Object Typing in O'Caml

Emmanuel Chailloux

http://www.pps.jussieu.fr/~emmanuel

Equipe Preuves, Programmes et Systèmes (CNRS UMR 7126)

University Paris 6 - France

Lisp Conference'2002 - San Francisco

# SUMMARY

1. O'Caml in few words
2. Object extension
3. `coca-ml` extension
4. Applications
   - Classes hierarchy
   - Design patterns
   - Persistence

# Objective Caml in practice

- One of the most popular ML dialect:
  - efficient code,
  - large set of general purpose and domain specific libraries,
  - automatic memory management,
  - used both for teaching (academy) and for writing high-tech applications (industry)
- Product of research results since 80's in: type theory, language design and implementation.
- Developped at INRIA (France).

# Objective Caml features

- Functional language + imperative extension,
- High-level datatypes + pattern-matching,
- Polymorphic + *implicit* typing:
  - strongly and static typed,
  - types are inferred,
  - types are polymorphic (the most general ones).
- Different programming styles (in a common typing framework):
  - Class based object oriented programming,
  - High-level modules (SML style)
  - *More recently*: labels and variants added.

# A Small Example

```
# let rec map f l =
  if (l == []) then []
  else (f   (List.hd l)) :: (map f (List.tl l));;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

`map` : $\forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow \alpha\, list \rightarrow \beta\, list$

```
# ( map (fun x -> x + 1) [1;2;3],
    map (fun x -> not x) [true; false]);;
- : int list * bool list = ([2; 3; 4], [false; true])
```

# Object extension

**It's not an object Language:**

- class declaration defines an object type and a constructor
- class instance has its object type
- object type containts method names and method types
- no access to instance variables, only via methods
- only static typing, no dynamic typing
- inheritance, late-binding
- abstract class, parametrized class, multiple inheritance
- subtyping is not inheritance
- inclusion polymorphism

# Syntax

**class structuration:**

**class** [**virtual**] name [ $p_1$ $p_2$ ... $p_n$ ] =

  **object** [ ( p ) ]

    **inherit** $\text{name}_c$ [ $p_i$ $p_j$ ]

    **constraint** typeexpr = typeexpr

    **val** [**mutable**] ident = expr

    **initializer** expr

    **method** [**private**] [**virtual**] $\text{name}_m$ = expr

  **end**

A class declaration computes : a type abbrevation for the types of objects AND a constructor function.

# A basic example : class p

```
class p  x_i y_i =
  object
    val mutable x = x_i
    val mutable y = y_i
    method gx = x
    method mv (a , b) = x <- a; y<- b
    method ts () = Printf.sprintf "(%d, %d)" x y
end
```

- type `p` : `type p = < gx :  int; mv :  (int*int) -> unit; ts :  unit -> string >`
- constructor : `int -> int -> p`

# Instances and methods

**creating instances : `new` primitive:**

```
# let p1 = new p 10 20;;
val p1 : point = <obj>
```

**Calling a method : sharp notation:**

```
# p1#get_x;;
- : int = 10
# p1#ts ();;
- : string = "( 10, 20)"
```

# Inheritance from p

```
class c_p x_i y_i c_i =
  object (self)
    inherit p x_i y_i as super
    val mutable c = c_i
    method gc = c
    method ts () = super#ts() ^ self#gc
```

- type `c_p` = `< gc : string; gx : int; mv : (int*int) -> unit; ts : unit -> string>`
- constructor : `int -> int -> c_p`

# Late binding

```
class nc_p x_i y_i c_i =
  object
   inherits c_p x_i y_i c_i
   method gc = "NO COLOR"
```

- `gc` is redefined but `ts` no.

```
# let p7 = new c_p 10 20 "blue";;
# let p8 = new n_c_p 10 20 "blue";;
# p7#ts();;
- : string = "(10, 20) blue"
# p8#ts();;
- : string = "(10, 20) NO COLOR"
```

# Abstract class

```
class virtual printable () =
  object(self)
    method virtual ts : unit -> string
    method print () = print_string (self#ts())
  end
```

- type `printable : <print : unit -> unit; ts : unit -> string>`
- no constructor

# Multiple inheritance

```
class pp x_i y_i =
  object
    inherit printable () as super1
    inherit p x y as super2
  end
```

- declaration order to bind methods
- distinguishing inherited methods by names of super-classes

# Parametrized class

- introduce parametric polymorphism in the object model

```
class ['a] queue () =  object
  val mutable q:'a list=[]
  method enq x = q <- q@[x]
  method deq = match q with
     [] -> failwith"Empty"  | h::r -> q<-r; h
end
```

- `type 'a queue = < deq :  'a; enq :  'a -> unit>`
- constructor : `unit -> 'a queue`

```
# let q = new queue ();;
# q#enq 3;;
# q;;    (* - : int queue = <obj> *)
```

# Objects and types

**open types:**

```
# let f o = (o#ts()) ^ "\n";;
val f : < ts : unit -> string; .. > -> string = <fun>
```

argument of `f` has an object type which containts a method `ts` :

```
# f (new p 10 20);;
# f (new c_p 10 20 "blue");;
```

**type inference:**

- no free type variables in a type (class) declaration,
- open types contain a free variable .. (row-polymorphism),
- no open types inside a method type.

# Recursive types

```
class p_eq x_i y_i =
  object (self :'a)
    inherit p x_i y_i
    method eq (p : 'a) = self#gx == p#gx
  end
```

- type `p_eq` : μ`'a.<eq : 'a -> bool; gx :int; mv : (int*int) -> unit; ts : unit -> string>`
- constructor : `int -> int -> p_eq`

# Using subtyping to cast object

- relation between types
- an object `o` of type `ot1` can be considerated as an object of type `ot2` iff
  `ot1` is a subtype of `ot2`
- (up)cast is explicit : `( o :  ot1 :> ot2)`
- no downcast

```
# let p12 = new c_p 10 20 "blue";;
val p12 : c_p = <obj>
# let p13 = (p12 : c_p  :> p);;
val p13 : p = <obj>
```

# Subtyping between object types

Let $t =< m_1 : \tau_1; \ldots\, m_n : \tau_n >$ and $t' =< m_1 : \sigma_1 \;; \ldots ;\; m_n : \sigma_n ; m_{n+1} : \sigma_{n+1} ; etc \ldots >$ we shall say that $t'$ is a subtype of $t$, denoted by $t' \leq t$, if and only if $\sigma_i \leq \tau_i$ for $i \in \{1, \ldots, n\}$.

**Subtyping of functional types.** Type $t' \rightarrow s'$ is a subtype of $t \rightarrow s$, denoted by $t' \rightarrow s' \leq t \rightarrow s$, if and only if

$$s' \leq s \text{ and } t \leq t'$$

The relation $s' \leq s$ is called **covariance**, and the relation $t \leq t'$ is called **contravariance**.

# Subtyping and inheritance

```
class printable_p_eq x_i y_i =
  object
    inherit printable () as super1
    inherit p_eq x_i y_i as super2
  end;;
```

- `printable_p_eq` is subtype of `printable`
- `printable_p_eq` is NOT subtype of `p_eq` :
  contravariance for the functional type of the `eq` method
  because the method `print` can be used in the body of method `eq`
  : `((new printable_p_eq 10 20 ) :> p_eq)#eq(new p_eq 10 20)` is dangerous and forbidden!!!

# Inclusion polymorphism

- subtyping + late-binding ⇒ inclusion polymorphism

```
# let q = new queue ();;
# q#enq(new p 10 20);;
# q#enq((new c_p 10 20 "blue") :> p);;
# q;;
- : p = <obj>
# q#deq#ts();;
- : string = "(10, 20)"
# q#deq#ts();;
- : string = "(10, 20)blue"
```

# Remarks

- object types are statically inferred : no ”method not found” exception but no overloading
- to call a method is less efficient that to apply a function : no optimization to detect a total application
- row polymorphism (record of methods with a row type variable) for object types : very close to subtype
- late-binding allows to modify behaviours of software components without sources
- it's an object extension : can be needed to use functional and object paradigms to solve a problem
- can be encapsulated in module declaration (see next section)

# coca-ml Extension (1)

Type constraints with dynamic typechecking!!!

**Motivations**

1. expressivity of the language
   - building classes hierarchy
   - using Design Patterns
2. Persistence
3. Interfacing between object languages

# coca-ml Extension (2)

To allow **cast** (mainly **downcast**) inside a classes hierarchy.

**Idea** : dynamic checking if construction class of an instance has inheritance and structural subtyping relations to the target class.

**Implementation** : programs transformation using `camlp4`

**Interest** : object paradigm

# Operator and Relations

- `cc(o)` :
- `c1` inherits from `c2`
- `c1` sub-inherits from `c2` iff
  `c1` inherits from `c2` AND `c1` ≤ `c1`

`cast o to c` is allowed iff `cc(o)` sub-inherits from `c`

# Syntax extension (1)

Predicates :

- `o` **of** `c` : `cc(o) = c`
- `o` **instanceof** `c` : `cc(o) = c` OR `cc(o)` inherits from `c`
- `o` **subinstanceof** `c` : `cc(o) = c` OR `cc(o)` sub-inherits from `c`

# Syntax extension (2)

Type constraints :

- **cast** `o` **to** `c` :
  - `o` is considerated as type `c` OR
  - an exception is raised if `o` sub-inherits from `c` is FALSE
- **upcast** `o` **from** `d` **to** `c` AND
  **downcast** `o` **from** `d` **to** `c`
  introduce some static typechecks

# Syntax extension (3)

Class declaration :

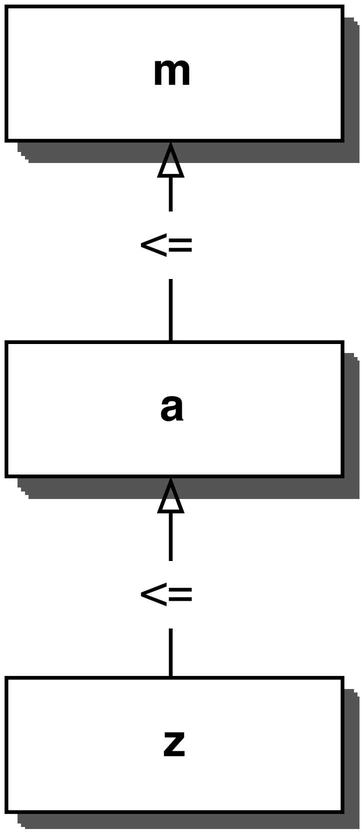
- **subinherit** `class_name [` $p_i$ $p_j$ `]`

  → inheritance and structural subtyping

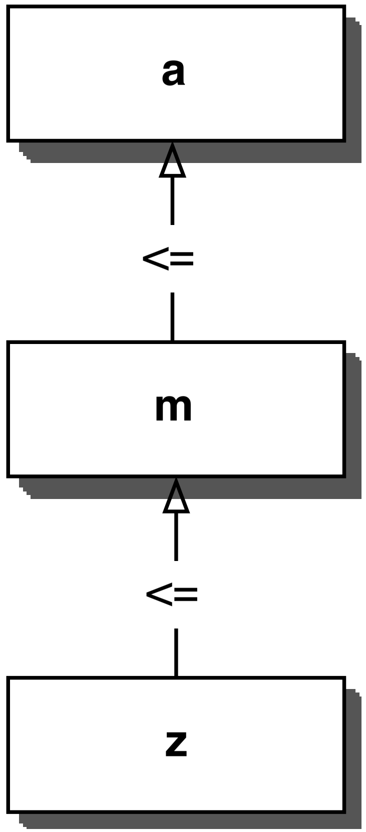
**Example** : `cast (cast (new z) to a) to m;;`

`a` sub-inherits of `m`




OK

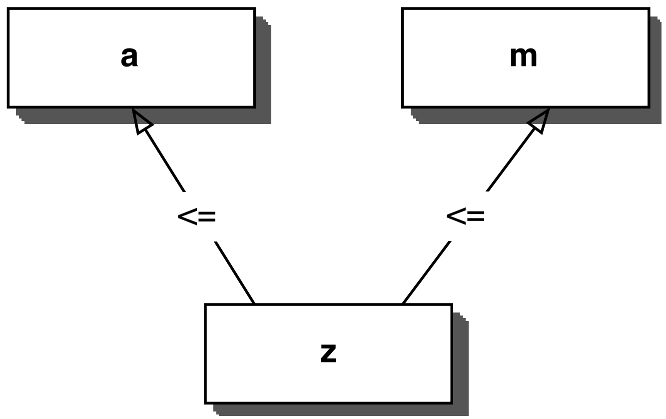
`m` sub-inherits of `a`




OK

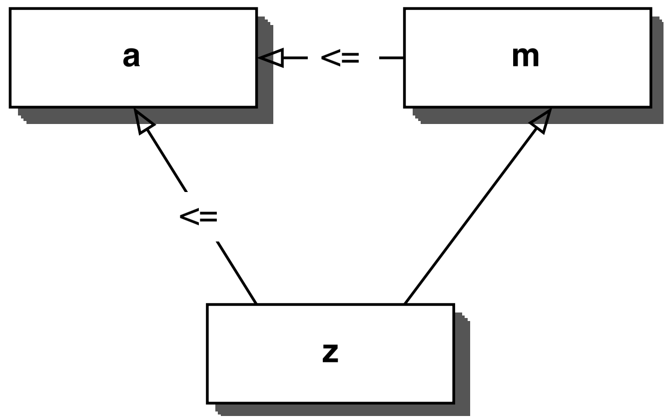
**Example** : `cast (cast (new z) to a) to m;;`

`m` and `a` are independant




OK

`z` is not a sub-type of `m`




CLASH

# Implementation (1)

- using `camlp4` : a tool for macro-expansion in O'Caml
- no types manipulation during evaluation

**a class declaration** generates :

- an unique ID for each class
- new methods for predicates
- a type constraint for sub-inheritance

# Implementation (2)

**predicates** and **cast** are transformed to :

- `expr` predicate `c` : appropriate method call to the object `o`
- `cast expr to c`

```
let o = expr in
  if o#check_subinstance_of(key_c)
  then ((Obj.magic o ) : c)
  else raise (Cast_failure loc ...)
```

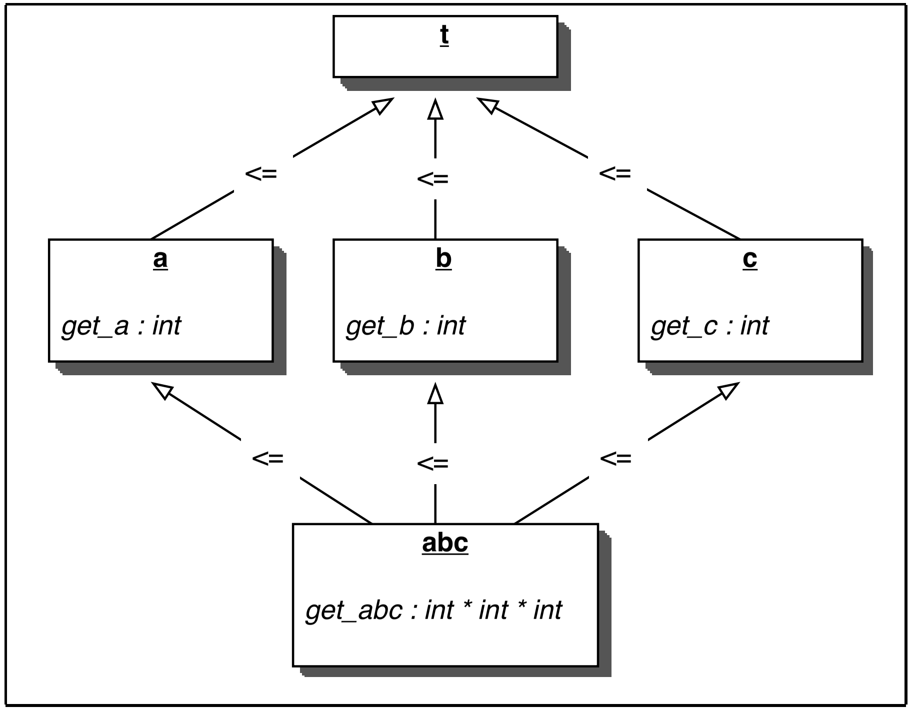
# Example

`z` inherits from `a` and `z` sub-inherits from `m` :

```
let ___z = (( Cast.make_id ()) : Cast.id );;

class z =
  object
    inherit a as ___super_0
    inherit m as ___super_1

    method ___check_instanceof_id (k : Cast.id) =
      (___z == k || ___super_1#___check_instanceof_id k)
                 || ___super_0#___check_instanceof_id k

    method ___check_subinstanceof_id (k : Cast.id) =
      (___z == k || ___super_1#___check_subinstanceof_id k)
```

```
    method ___check_subclass (k1 : Cast.id) (k2 : Cast.id) =
      if (___z == k1) then
        if (k1 == k2) then true
        else ___super_1#___check_subclass k2 k2
      else ___super_1#_check_subclass k1 k2
...
end;

fun ___x -> (___x : z :> m);;
```

# **Example** : multiple inheritance




```
cast (cast (new z) to a) to b;;
```

OR

```
upcast (  downcast  (upcast (new z) from z to a)
          from a to z ) from z to b
```

## Compatibility, cost and limitation

**Syntax** : keywords, added methods

**operator :>** :
is allowed by it doesn't check est autorisé mais ne vérifie pas l'appartenance à la hiérarchie de clas ses **Cost**

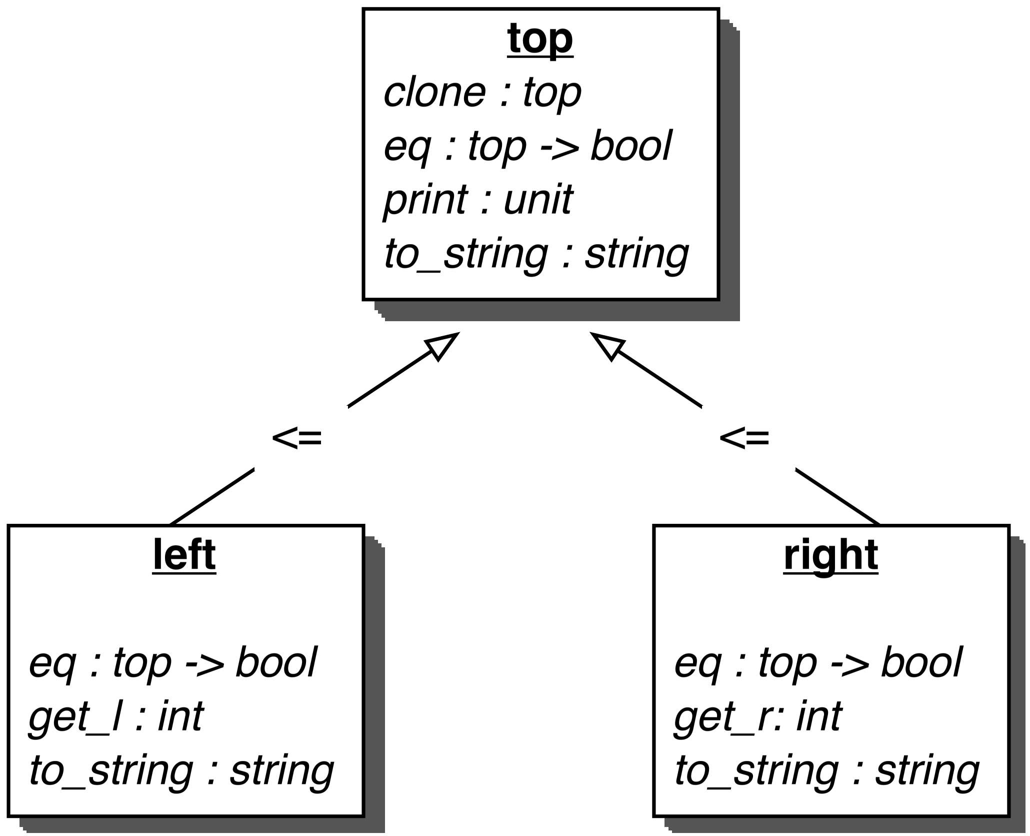
- very low if you don(t use it
- depends of the hierarchy deep + partial application (methods)

**Limitation**

- can't be used for parametric classes :

```
cast (
  cast (new alpha_pile 1) to top )
to float alpha_pile
```

# Classes hierarchy : the class `top`



## main class : top

```
class virtual top =
  object(self)
    method virtual to_string :  string
    method print = print_string (self#to_string)
    method clone = Oo.copy(self)
    method eq (a : top) = true
 end;;
```

# subclass : left

```
class left (x : int) =
  object (self  )
  subinherit top
  val l = x
  method get_l = l
  method to_string = string_of_int l
  method eq ( x : top) =
    let xl = downcast x from top to left in
        l = (xl#get_l)
end;;
```

## Program

```
let l = new left 10;;
let l2 = new left 10;;
let r =  new right 10;;
if l#eq(cast l2  to  top) then print_string "OK"
else print_string "PB";;
try
 ignore(l#eq (cast r to  top));
  print_string "PB"
with Cast.Cast_failure _ -> print_string "OK";;
```

# Persistence

Main problems :

- for the values : read-write of data structures, share or copy, circular structures and functional values
- for the types : guaranteeing the type when reloading

# O'Caml Marshalling

The two main functions are :

- `to_string` : $\forall\alpha.\alpha \rightarrow extern\_flaglist \rightarrow string$
- `from_string` : $\forall\alpha.string \rightarrow int \rightarrow \alpha$

NO guaranteeing the type when reloading values!!!

# Danger

```
# let magic_copy a =
    let s = Marshal.to_string a [Marshal.Closures]
    in Marshal.from_string s 0;;
val magic_copy : 'a -> 'b = <fun>
```

→ benefit of the static typechecking is lost :

```
#  (magic_copy 3 : float) +. 3.1;;
Segmentation fault
```

# Persistence using `coca-ml`

**Idea** : To resolve typing problems with the dynamic object cast.

Each class auto-sub-inherits from `serialize` class ancestor
⇒ to downcast a reread object to its original class or to an intermediate class between this one to `serialize`.

# Implementation

- to define an unique ID for each class : MD5 digest from AST;
- to find the Methods Table (unique for each class) : partial application of the constructor;
- to build a global hash-table (ID, MT)
- to linearize circular structures;

# New module : `Dumpto`

- `to_string` : $\forall \alpha . \alpha \rightarrow string$
- `from_string` : $string \rightarrow serialize$

## Example

```
# let o = let s = DumpTo.to_string l
   in  DumpTo.from_string  s;;
o : serialize = <obj>
# let t = cast o to top;;
t : top = <obj>
# let lt = cast o to left;;
lt : left = <obj>
# lt#eq(t);;
- : bool = true
# let r = cast o to right;;
# let r = cast o to right;;
Uncaugth exception : ...
```

# Limitation

1. implementation choice : we do not treat functional variable instances;
2. restriction for a class declaration to find safely its methods table;
3. restriction from parametric polymorphism;

→ only for the object part : no test ti check correct link with global variable

# Related work

- Extended MOBY (Fisher - Reppy) : inheritance-based subtyping but no downcast
- Dynamics (Leroy - Mauny) : type-case pattern matching only exact type
- Types reconstruction (Aditya - Naro) : needing added types informations for application ( side-effects, exceptions)
- Structral persistence (Furuse - Weis) : works only on value representations

# Conclusion

`coca-ml` opens the Pandora Box (dynamic typechecking)

BUT :

- casts are explicit
- no good solution when types are not generated by its own program