Objects and classes vs. modules in Objective Caml

Xavier Leroy

INRIA Rocquencourt & Trusted Logic

Outline

Objective Caml supports both

- class-based object-oriented programming;
- an SML-style module system.

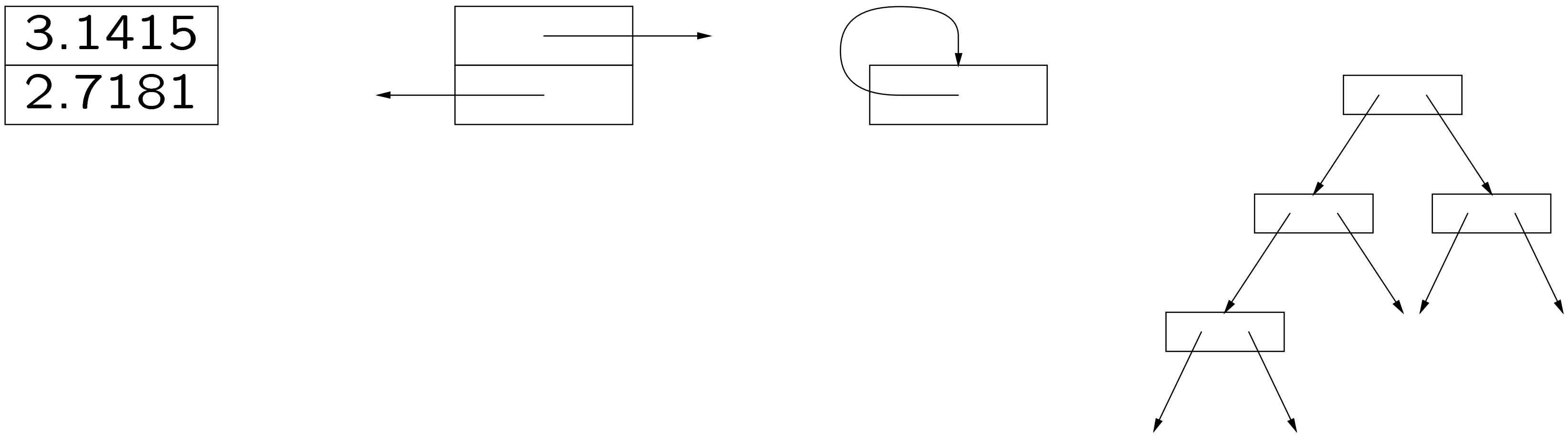
A good opportunity to compare those two code structuring mechanisms:

- potential overlaps;
- unique strengths of each;
- some problems that require the combination of both.

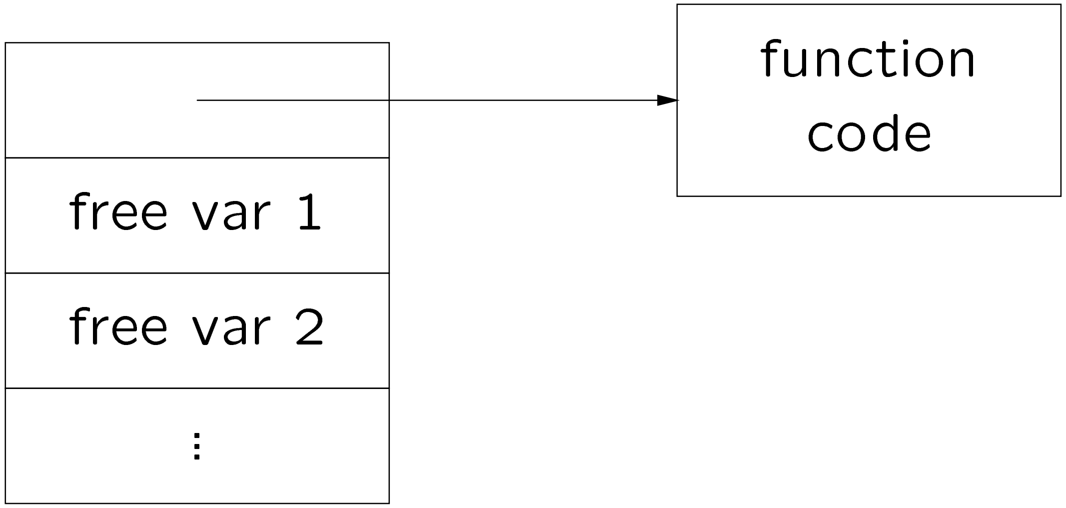
Part 1

Objects

In the beginning was the memory heap

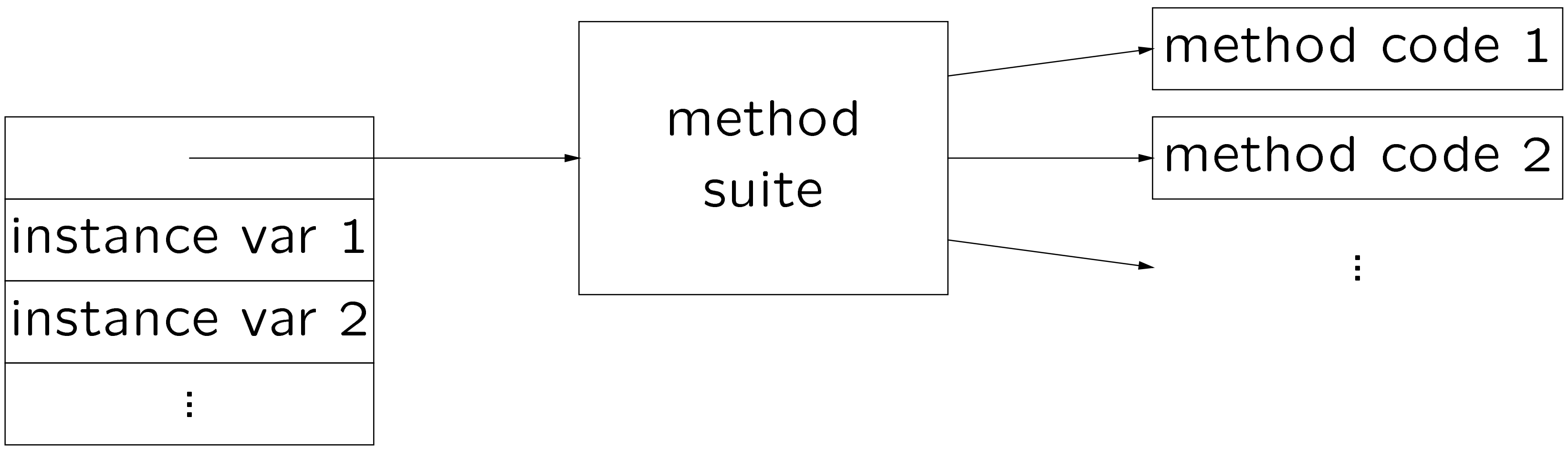


Closures



Self-application semantics: $[\![f(a)]\!] = [\![f]\!].\mathtt{code}([\![f]\!], [\![a]\!])$

Objects



Self-application semantics: $[\![o\#m(a)]\!] = [\![o]\!].\mathtt{methsuite}(m)([\![o]\!], [\![a]\!])$

Objects vs. closures

- Objects = generalized closures with multiple entry points.
- Closures = degenerate objects with only one `apply` method.

Specific to objects:

- Extra methods don't matter → subtyping.
- Incremental construction of the method suite → inheritance; method overriding; late binding; open recursion.

Typing objects

Static type-checking of objects is (mostly) well understood in the context of explicit typing.

Subtyping between object types:

- width subtyping: $\langle m : A; m' : B\rangle <: \langle m : A\rangle$
- depth subtyping: $\langle m : A\rangle <: \langle m : A'\rangle$ if $A <: A'$.

The big issue for us is to combine type inference with subtyping — more exactly, with implicit subsumption (if $A <: A'$ then any term of type A can be used with type A').

Type inference with subtyping

Inference \ Subsumption	none	explicit	implicit
full	ML, Haskell	Objective Caml	constraint-based inference
partial	proof assistants		local inference
none	boring languages		Java, Modula-3

The OCaml approach: full inference, row polymorphism

(Wand, Rémy for extensible records; Rémy, Vouillon, Davies for objects.)

Rows list the methods of an object along with their type, and possibly a row variable at the end to allow extension:

- Closed rows: $\langle m : \mathtt{int}; p : \mathtt{string}\rangle$
  (the type of an object value)

- Open rows: $\langle m : \mathtt{int}; p : \mathtt{string}; \rho\rangle$
  (the type of a function or method parameter)

Hindley-Milner type inference easily extends to rows and row polymorphism, leading to a complete type inference algorithm.

The OCaml approach: explicit subsumption

Row polymorphism is as good as subtyping in many practical situations:

```
let f obj = obj#m + 1
     f : ∀ρ. <m:int; ρ> -> int
```

Assuming `obj1 : <m:int; p:string>` and `obj2: <b:bool; m:int>`,

```
(f obj1, f obj2)
     - : int * int
```

Row polymorphism is not subtyping. In particular, the former does not allow building heterogeneous data structures such as heterogeneous lists:

```
[ obj1; obj2 ]
     Cannot unify <m:int; p:string> with <b:bool; m:int>
```

Subtyping is available as an explicit coercion operator $(a : \tau_1 :> \tau_2)$ or $(a :> \tau_2)$.

```
[ (obj1 :> <m:int>); (obj2 :> <m:int>) ]
- : <m:int> list
```

Part 2

Classes

Classes

Few OO languages allow the programmer to build an object by listing directly its methods and instance variables.

Most, including OCaml, introduce a notion of *classes* to act as object templates:

- Classes group together instance variable declarations and method definitions or declarations.
- Objects are generated from classes (the `new` operator).

Running example: unified input/output

A class encapsulating output on the terminal:

```
class standard_output =
  object
    method put_char c = print_char c
    method put_string s = print_string s
    method put_int n = print_int n
  end

let stdout = new standard_output
     val stdout : standard_output = <obj>
```

Note: the type `standard_output` is an automatically-generated abbreviation for the type of the class instances, i.e.

```
  < put_char: char -> unit; put_string: string -> unit; put_int: int -> unit >
```

Inheritance and method overriding

Classes support a form of incremental programming via inheritance (re-use of methods from another class) and method overriding (re-definition of inherited methods).

A common scenario:

- We build an (incomplete) class defining some default implementations of methods, often in terms of calls to other methods.

- We then use inheritance between classes to reuse the default implementations in a derived class.

- In that derived class, we can add methods with new functionality

- We can also redefine methods for which we have a better implementation than the default one.

Running example: writers and file output

```
class virtual writer =
  object (self)
    method virtual put_char : char -> unit
    method put_string s =
      for i = 0 to String.length s - 1 do self#put_char s.[i] done
    method put_int n = self#put_string (string_of_int n)
  end

class file_output filename =
  object
    inherit writer
    val chan = open_out filename
    method put_char c = output_char chan c
    (* Override put_string with a more efficient implementation *)
    method put_string s = output_string chan s
    (* Keep the default put_int implementation, it's efficient enough *)
    (* Define some additional methods *)
    method seek pos = seek_out chan pos
    method close = close_out chan
  end
```

Late binding (open recursion)

Cross-references between methods of a class are (roughly speaking) determined when we do `new`, but not yet when the classes are defined.

Thus, inheritance naturally allows mutually recursive definitions where members of the recursion can be changed later.

```
class c =
  object (self)
    method even n = n=0 || self#odd(n-1)
    method odd n = n=1 || self#even(n-1)
  end

class d =
  object
    inherit c
    method even n = (n mod 2) = 0
  end

let o = new d; o#odd 1000000001
(* terminates immediately *)
```

```
let rec even n =
  n=0 || odd(n-1)
and odd n =
  n=1 || even(n-1)

let even n =
  (n mod 2) = 0

odd 1000000001
(* makes 1 billion iterations *)
```

Part 3

Modules

ML modules for dummies

Base modules (structures):
collections of definitions

```
struct
  type t = int
  let f x = x + 1
  module M = struct
               let z = 2
             end
end
```

Their types (signatures):
collections of type specifications

```
sig
  type t
  val f : t -> t
  module M : sig
               val z : t
             end
end
```

```
functor(X : SIG) ->
  struct...end
```

Parameterized modules (functors):
functions from modules to modules

```
F(A)
```

Linking of modules:
functor application

An example of functor: polynomials over a ring

```
module type RING =
  sig
    type t
    val zero: t
    val one: t
    val plus: t->t->t
    val neg: t->t
    val mult: t->t->t
  end
```

```
module type POLYNOMIAL =
  sig
    module R: RING
    type t
    val zero: t
    val one: t
    val monomial: R.t->int->t
    val plus: t->t->t
    ...
  end
```

```
module SparsePoly(Ring: RING) : (POLYNOMIAL with R.t = Ring.t) =
  struct
    module R = Ring
    type t = (R.t * int) list
    let zero = []
    ...
  end
```

Main uses of the module system

As a general mechanism for environment manipulation:

- Structures and the dot notation = name space control

- Functors = mass parameterization

```
                                   module F(X: sig ..g..h..k.. end) = struct
let f1 g h k = ...                   let f1 = ...
...                                  ...
let fn g h k = ...                   let fn = ...
                                   end
(* N partial applications *)       (* 1 functor application *)
```

To model abstract data types as data types interpreted by operations:

- Structures = simple A.D.T.

- Functors = A.D.T. parameterized by another A.D.T.

Part 4

Classes vs. modules

Software evolution

Classes and modules are not really needed for closed, non-evolving programs
(except if required to support separate compilation).

Classes and modules become significant for software that

- should remain extensible over time

  (ex: adding support for a new target processor in a compiler);

- is intended as a framework or set of components to be used in larger programs

  (ex: libraries, toolkits, . . .).

Myth: OO programs are easier to extend

Consider a type of "things" with several kinds of things and several operations over things.

	FP approach: One function per operation, defined by pattern-matching on things	OO approach: One class per kind of things, one method per operation
Adding a new kind of thing	Must edit all functions, adding a new case to every pattern-matching	Add one class, other classes unchanged
Adding a new operation over things	Add one function, other functions unchanged	Must edit all classes, adding a new method to every class

A case of overlap: simple algebraic data types

A.D.T. that are mutable and whose operations have only one parameter of the A.D.T. type can be expressed equally well as modules or as classes.

```
module Stack:                              class ['a] stack:
  sig                                        object
    type 'a t
    val create: unit -> 'a t
    val push: 'a t -> 'a -> unit               method push: 'a -> unit
    val pop: 'a t -> 'a                        method pop: 'a
  end                                        end
```

```
   'a Stack.t  ↔  'a stack
Stack.create()  ↔  new stack
```

The choice is purely a matter of style.

Inheritance in the module system

Extension structures and functors can express simple cases of inheritance:

```
module ExtStack(S:Stack) =
  struct
    type t = S.t
    let create = S.create
    let push = S.push
    let pop = S.pop
    let rec length s =
      try
        let top = S.pop s in
        let len = length s in
        S.push s x; len + 1
      with Empty ->
        0
  end
```

```
class ['a] extstack =
  object(self)
    inherit ['a] stack



    method length =
      try
        let top = self#pop in
        let len = self#length in
        self#push top; len + 1
      with Empty ->
        0
  end
```

The module-based solution breaks if we override a function called by other functions (closed recursion vs. open recursion).

Example: instruction selection in the OCaml compiler

Certain passes of a code generator (instruction selection, reloading, scheduling) depend on the target processors in subtle ways:

- Large parts of those passes are common to all (or most) target processors, and should be written once and for all.

  (Ex: instruction selection for an `if...then...else`)

- The processor description should be able to provide its own implementation for parts of those passes, to take advantage of processor idioms.

  (Ex: recognition of addressing modes and of combined instructions such as “multiply-add”.)

A solution based on modules

In the processor description:

```
module PowerPC = struct
  module Selection = struct
    let select_expr op args =
      match (op, args) with
        (* Recognize processor idioms *)
        (Add, [Mult(a1, a2); a3] -> Multadd(a1,a2,a3)
      | ...
        (* Request default selection for all other expressions *)
      | _ -> raise Use_default
  ...
```

In the shared code:

```
module Selection(P:Processor) = struct
  let rec select_expr op args =
    try
      (* See if the processor description recognizes this operation *)
      P.Selection.select_expr op args
    with Use_default ->
      (* Perform default instruction selection *)
      match (op, args) with ...
```

A solution based on classes and inheritance

In the shared code:

```
class default_selector = object
  method select_expr op args =
    (* Perform default instruction selection *)
    match (op, args) with ...
```

In the processor description:

```
class selector = object
  inherit default_selector as super
  method select_expr op args =
    match (op, args) with
      (* Recognize processor idioms *)
      (Add, [Mult(a1, a2); a3] -> Multadd(a1,a2,a3)
    | ...
      (* Use default selector for all other expressions *)
    | _ -> super#select_expr op args
```

Comparing the two solutions

The solution based on modules:

- It is very clear what each processor description must provide.
- Lots of trivial functions `raise Use_default` in processor descriptions.
- Making a generic selection operation overridable entails editing all existing processor descriptions.

The solution based on inheritance:

- Existing processor descriptions need not be changed if we add a new generic selection method.
- Makes it easier to support one processor with plenty of processor-specific gunk (IA32) and many processors with few (RISCs).
- It is less clear what each processor description must provide.

Example: computer algebra

(The FOC project, Boulmé, Hardin, Rioboo, Paris 6.)

A large library of algebraic structures a la Axiom. Challenges:

- Complex relationship between structures.
- Many operations have a default implementation applicable to many structures, but must be redefined in particular structures.
- Some algorithms are quite complex; their code cannot sensibly be duplicated.
- Representation hiding is crucial (an element of $\mathbb{Z}/n\mathbb{Z}[X]$ is not a list).
- Execution speed matters.

First approach: modules only

An algebraic structure = an abtract type `t` + operations over it.

```
module type Ring = sig
  type t
  val zero: t
  val one: t
  val plus: t -> t -> t
  val mult: t -> t -> t
  val opp: t -> t
  val sub: t -> t -> t
  ...
end
```

```
module Z2Z: Ring = struct
  type t = int
  let zero = 0
  let one = 1
  let plus = lxor
  let mult = land
  let opp x = x
  let sub x y = plus x (opp y)
  ...
end
```

Problem: hard to derive a structure from another one by addition or overriding of operations (no late binding).

Second approach: objects only

An element of an algebraic structure =
one object carrying the operations of the structure.

```
class z2z =
  object(self:'a)
    val r = 0
    method rep = r
    method zero = {<r=0>}
    method one = {<r=1>}
    method plus (x:'a) = {<r=r lxor x#rep>}
    method mult (x:'a) = {<r=r land x#rep>}
    method opp = self
    method sub (x:'a) = self#plus (x#opp)
  end
```

Good support for inheritance, but:

- Operations are artificially attached to elements of the structure (is it `x#plus y` or `y#plus x`?)
- Binary methods force the representation type to be exposed (the `rep` method).
- Insufficient type abstraction (elements of $\mathbb{Z}/2\mathbb{Z}$ and $\mathbb{Z}/3\mathbb{Z}$ have compatible types).

Mixed approach: classes + modules

An algebraic structure =

one parametric class for inheritance of operations

+ one abstract type for the elements

+ one object for the operations.

```
class virtual ['a] ring_ops =
  object (self)
    method virtual zero: 'a      method virtual plus: 'a->'a->'a
    method virtual one: 'a       method virtual mult: 'a->'a->'a
    method virtual opp: 'a->'a
    method sub x y = self#plus x (self#opp y)
  end

module type Ring = sig
  type t
  val ops: t ring_ops
end

class z2z_ops =
  object
    inherit [int] ring_ops
    method zero = 0              method plus = lxor
    method one = 1               method mult = land
    method opp x = x
  end

module Z2Z : Ring = struct
  type t = int
  let ops = new z2z_ops
end
```

Part 5

Conclusions

Modules vs. classes: a summary

Things that work better with modules:

- Clear distinction between the values and the operations over them (better for n-ary operations).
- Stronger representation hiding.
- Functors as a general parameterization mechanism.
- Safety at the expense of flexibility.

Things that work better with objects and classes:

- Subtyping and row polymorphism at the level of values.
- Incremental programming via inheritance (open recursion, default implementations that can be overriden later).
- Flexibility at the expense of safety.

Alternatives worth investigating

An alternative to ML-style modules: mixin modules

(Bracha; Duggan; Ancona and Zucca; Flatt and Felleisen)

- Support for open recursion and overriding.
- Support for mutually-recursive modules.
- Believed to be able to encode functors.

An alternative to class-based OO programming: multi-methods

(CLOS; Chambers; Castagna; Bourdoncle et al)

- Separate object representations and the methods that work over them.
- Better support for n-ary operations.

Bonus feature

A more sociological perspective

Modules and A.D.T.	Classes and objects
Protect against programmers	Trust programmers
Pessimistic mindset	Optimistic mindset
Educated terminology (signatures, structures, functors)	Common words (objects, classes, methods, to inherit)
Mathematical perspective (operations over abstract types; algebraic laws)	Physical/anthropomorphic perspective (send messages asking objects to do things)
Elitist discourse	Populist discourse
...	...