Object Oriented Features

OCaml PRO – Benjamin Canou

March 9-13 2015

Outline

Objects

Defining and calling objects
Object types
Instance variables
Object closures
Recursion

Objects are first class OCaml constructions.

- Their definition is a first class expression.
- Syntax: `object ... end`
- They define a set of methods.
- Syntax: `method f x y z = body`
- Methods can take no argument, unlike functions (to reserve for accessors).
- Calling syntax: `obj#f x y z`

Defining and calling objects

Through this lecture, we'll define imperative streams, using objects.

We start with a flatline stream.

```
let zero = object
  method next () = 0
  method reset () = ()
end
```

To test our streams, we write a function:

```
let test obj =
  print_int (obj#next ()) ;
  print_int (obj#next ()) ;
  print_int (obj#next ())
```

Object types 1 / 2

The `zero` object has type:

```
val zero : < next : unit -> int; reset : unit -> unit >
```

The type of an object is the map between its public methods names and types.

- A method has only one type: no overloading in OCaml.
- Syntax: `< name : type ; ... ; name : type >`.
- Object types are first class types, and can be aliased.
- Example:
  `type stream = < next : unit -> int ; reset : unit -> unit >`.

This is structural object typing.

- As opposed to nominal typing (the type of an object is the name of its class).
- The strict, static interpretation of duck typing.

The `test` function has type:

```
val test : < next : unit -> int; .. > -> unit
```

Reading:

- The parameter must be an object (`< (* ... *) >` type syntax).
- It must have a method `next`.
- Whose type must be `unit -> int`.
- And it can have any other method (`< (* ... *) ; .. >` syntax).

This is structural subtyping.

- An object is a subtype of another if its structure is included.
- As opposed to nominal subtyping where the relation is the classs hierarchy.
- Similar to the relations between module interfaces.
- The supertype of all objects is `< .. >`.

Instance variables

Objects can have instance variables.

- Syntax: `val name = value`.
- They must be initiazed, and can be declared mutable.
- Syntax: `val mutable name = value`.

```
let ints = object
  val mutable counter = 0
  method next () =
    let prev = counter in
    counter <- counter + 1 ;
    prev
  method reset () =
    counter <- 0
end
```

They never appear in the type.

```
val ints : < next : unit -> int; reset : unit -> unit >
```

Objects can be initialized using their definition environment.

```
let ints_from n = object
  val mutable counter = n
  method next () =
    let prev = counter in
    counter <- counter + 1 ;
    prev
  method reset () =
    counter <- n
end
```

As functions, they actually embed their necessary environment.

```
exception End_of_stream

let ints_between n m = object
  val mutable counter = n
  method next () =
    if counter = m then raise End_of_stream ;
    let prev = counter in
    counter <- counter + 1 ;
    prev
  method reset () =
    counter <- n
end
```

The object can refer to itself.
No `this` keyword, the naming is explicit.

```
let between n m = object (self)
  val mutable counter = n
  method next () =
    if counter = m then raise End_of_stream ;
    let prev = counter in
    counter <- counter + 1;
    prev
  method reset () =
    counter <- n
  method to_list () =
    match self#next () with
    | head -> head :: self#to_list ()
    | exception End_of_stream -> []
end
```

Objects & Polymorphism

A polymorphic polymorphism
Parametric object types
Polymorphic methods
Recursive object types & polymorphism
Row polymorphism
Coercions

A polymorphic polymorphism

Four kinds of polymorphism with objects in OCaml.

1. Usual object polymorphism (inheritance).
2. Parametric object types (aka. generics / templates).
3. Parametric method types (aka generic methods).
4. Row polymorphism (aka. duck typing).

First kind of polymorphism: object level parametric polymorphism.

We generalize `between_ints` by abstracting on `+ 1`.

```
let between n m succ = object
  val mutable counter = n
  method next () =
    if counter = m then raise End_of_stream ;
    let prev = counter in
    counter <- succ counter ;
    prev
  method reset () =
    counter <- n
end
```

Parametric object types 2 / 2

The type of `between` is:

```
val between :
  'a -> 'a -> ('a -> 'a) ->
  < next : unit -> 'a ;
    reset : unit -> unit >
```

We can name this type as a polymorphic type alias.

```
type 'a stream =
  < next : unit -> 'a ;
    reset : unit -> unit >
```

So we can rewrite the type of `between`.

```
val between : 'a -> 'a -> ('a -> 'a) -> 'a stream
```

Polymorphic methods

Second kind of polymorphism: method level parametric polymorphism.-

A method can be polymorphic, but this requires an annotation.

```
type 'a stream =
  < next : unit -> 'a ;
    reset : unit -> unit ;
    map_next : 'b. ('a -> 'b) -> unit -> 'b >
let rec between n m succ : 'a stream = object (self)
  (* ... *)
  method map_next
    : 'b. ('a -> 'b) -> unit -> 'b
    = fun f () -> f (self # next ())
end
```

This is because OCaml cannot guess if the type variable should be quantified for the type or the method only.

Recursive object types & polymorphism

We add a dup method that produces a new stream from its current state.

```
let rec between n m succ = object (self)
  val mutable counter = n
  method next () = (* ... *)
  method reset () = (* ... *)
  method to_list () = (* ... *)
  method dup () =
    between counter m succ
end
```

The infered type of `between` is

```
val between :
  'a -> 'a -> ('a -> 'a) ->
  (< next : unit -> 'a ;
     reset : unit -> unit ;
     to_list : unit -> 'a list ;
     dup : unit -> 'b > as 'b) =
```

The keyword `as` names the object type as an unexported variable `'b`.
The `dup` method returns this 'b.

Recursive object types & polymorphism 3 / 5

For clarity, we will use a recursive object type alias.
The intermediate variable can be replaced by a type recursion.

```
type 'a stream =
  < next : unit -> 'a ;
    reset : unit -> unit ;
    to_list : unit -> 'a list ;
    dup : unit -> 'a stream >
```

And adding the annotation:

```
let rec between n m succ : 'a stream = object (* ... *) end
```

We get a clearer type for `between` :

```
val between : 'a -> 'a -> ('a -> 'a) -> 'a stream
```

Type recursion is also useful to describe binary methods.

```
type 'a stream = < (* ... *) ; eq : 'a stream -> bool >
let rec between n m succ : 'a stream = object (self)
  (* ... *)
  method eq oth =
    let rec eq so oo =
      match so#next () with
      | sh ->
        begin match oo#next () with
          | oh -> sh = oh && eq so oo
          | exception End_of_stream -> false end
      | exception End_of_stream ->
        begin match oo#next () with
          | _ -> false
          | exception End_of_stream -> true end
    in eq (self#dup ()) (oth#dup ())
end
```

Note: the type of eq is actually too restrictive.

Known limitation: recursion must preserve the parameters.
The following is forbidden:

```
type 'a stream =
  < (* ... *) ; map : 'b. ('a -> 'b) -> 'b stream >
```

This kind of combinators must be implemented outside of the object.

```
let rec map
  : ('a -> 'b) -> 'a stream -> 'b stream
  = fun conv o -> object (self)
    method next () = conv (o#next ())
    method reset () = o#reset ()
    method to_list () = List.map conv (o#to_list ())
    method dup () = map conv (o#dup ())
    method eq oth = oth#eq self
  end
```

Third kind of polymorphism: polymorphic object types.

As we've seen with the `test` function, OCaml has structural subtyping.
An object type `< print : unit -> unit ; .. >`

- Reads: an onject with a method `print` and any other.
- We say it is an open object type.

A simple type alias cannot be bound to an open object type.

```
type printable = < print : unit -> unit ; ..>
```

Leads to an error:

```
Error: A type variable is unbound in this type declaration.
       In type < print : unit -> unit; .. >
       as 'a the variable 'a is unbound
```

Because `..` is a polymorphic part of the type.
It is called a row type variable.
As any variable exported by the right of an alias, it must appear on the left.

We can use the `as` construct.

```
type 'a printable = (< print : unit -> unit ; ..> as 'a)
```

To capture the row type variable in a variable `'a`.

Type variables capturing object polymorphism can be used recursively.

```
type 'a dupable = (< dup : unit -> 'a ; ..> as 'a)
let dup (o : 'a dupable) : 'a dupable = o#dup ()
```

Specifying that dup

- takes any dupable object ;
- returns one of the exact same type.

Structural subsumption (upcasting) is not implicit in OCaml.
A closed object type is not automatically coerced to a subtype when needed.

It is done:

- When calling a polymorphic function
  (instanciating the polymorphic type of the function to the subtype).
- By the programmer with `(value :> desired subtype)`
  (erasing the subtype specific methods).

Coercions 2 / 4

Example: a list of compatible objects

We define objects that have a common subtype `valuable`, and try to build a list.

```
type valuable = < value : int >
let valuable_b = object method value = 2 method b = () end
let valuable_c = object method value = 3 method c = () end
let valuables = valuable_b :: valuable_c :: []
```

We get error'd with

```
Error: This expression has type < c : unit; value : int >
       but an expression was expected
       of type < b : unit; value : int >
       The second object type has no method c
```

If we coerce them both, everything is Ok.

```
type valuable = < value : int >
let valuable_b = object method value = 2 method b = () end
let valuable_c = object method value = 3 method c = () end
let valuables =
 (valuable_b :> valuable) :: (valuable_c :> valuable) :: []
```

This means the elements of the list are of type `valuable`.
They cannot be returned to their original type: no downcasting in OCaml.

Coercions

Coercion is also used bakwards by the inference.

So we can give clean types to the previous examples in a simpler way.

```
type printable = < print : unit -> unit>
type dupable = (< dup : unit -> 'a > as 'a)
let print o = (o :> printable)#print ()
let dup o = (o :> dupable)#dup ()
```

Or our list example, replacing `::` by a specialzed `@::` :

```
let (@::) obj list = (obj :> valuable) :: list
let valuables = valuable_b @:: valuable_c @:: []
```

Classes

Classes or immediate objects ?
Class definitions in interfaces
Class type definitions
Parametric classes

Compared to immediate objects, classes add:

- A factorized syntax for object type definition and implementation.
  The name of the class is the one of both its constructor and type.
- Inheritance !
- A bit more annotations...

Syntax:

- Definition:
  `class name args = object (* ... *) end`
- With an environment:
  `class name args = let (* ... *) in object (* ... *) end`
- Instanciation: `new name args`

The basic example revisited:

```
class zero = object
  method next () = 0
  method reset () = ()
end
let z = new zero
```

With arguments:

```
class ints_between n m = object
  val mutable counter = n
  method next () =
    if counter = m then raise End_of_stream ;
    let prev = counter in
    counter <- counter + 1 ;
    prev
  method reset () =
    counter <- n
end
```

Class definitions in interfaces

A class definition appears in the interface.

- With its name, argument types and class-type
- Syntax: `class name : args -> type`

The class-type format is:

- Syntax: `object (* elements *) end`
- The list of methods with their types.
  `method f : x -> y -> z`
- The list of variables with their types
  `val x : t`, `val mutable x : t`

Class definitions in interfaces 2 / 2

The previous definition appears as:

```
class ints_between
  : int -> int ->
    object
      val mutable counter : int
      method next : unit -> int
      method reset : unit -> unit
    end
```

Class types can be named (similarily to module signatures).

- Syntax: `class type name = object (* ... *) end`
- Defines an object type of the same name.
- A class definition defines a class type of the same name.
  When you define a class, you also define a class type and a type.
- The class type of a class can be forced with an annotation.
- Variables can be hidden, but not methods.

Example, rewritten:

```
class type stream = object
  method next : unit -> int
  method reset : unit -> unit
end
class ints_between n m : stream = object (* ... *) end
```

In the interface:

```
class type stream = object (* ... *) end
class ints_between : int -> int -> stream
```

Class type definitions

Note: it's still not nominal typing.

```
class circle = object method draw () = () end
class square = object method draw () = () end
let f : circle -> unit = fun s -> s#draw ()
let () = f (new square) (* OK *)
```

The type system only knows about the structure.
Class type names just build type aliases for the type system.
No RTTI, `typeof`, `instanceof`, etc.

As with immediate objects, classes can have a type parameter.

- Definition syntax:
  `class ['a] name args = object (* ... *) end`
- Defines a polymorphic class type:
  `class type ['a] name = object (* ... *) end`
- And a polymorphic type: `type 'a name = < (* ... *) >`

Example, rewritten:

```
class type ['a] stream = object
    method dup : unit -> 'a stream
    method eq : 'a stream -> bool
    method next : unit -> 'a
    method reset : unit -> unit
    method to_list : unit -> 'a list
end
class ['a] between n m succ : ['a] stream = object
  val mutable counter : 'a = n
  (* ... *)
end
```

In the interface

```
class ['a] between
  : 'a -> 'a -> ('a -> 'a) -> ['a] stream
```

Object polymorphism

Inheritance
Private methods
Virtual methods and virtual classes

Syntax:

- Basic: `inherit mother args`
- No `super` in OCaml.
- Named: `inherit mother args as mom`

Multiple inheritance is allowed.

Child classes can:

- Access and update instance variables.
- Define additional methods and instance variables.
- Call the super methods (`mom#f x y`).
- Redefine methods using `method!` .
- Hide variables using `val!` .
- Only access what is visible in the class type of the mother.

Exemple: rewindable streams.

```
exception Start_of_stream

class ['a] rewindable_between n m succ pred = object (self)
  inherit ['a] between n m succ
  method rewind =
    if counter = n then
      raise Start_of_stream ;
    counter <- pred n
end
```

Private methods

A class can define methods that are only callable from the same object.

- Not from objects of the same class.
- Also in the code of a child class.
- Private methods can be made public in child classes.
- Syntax: `method private f args = body`

Private methods can be hidden by class type interfaces.

- As instance variables.
- Unlike public methods.

Exemple: private checked setter.

```
exception End_of_stream
exception Start_of_stream
class ['a] between n m succ = object (self)
  val mutable counter : 'a = n
  method private set_counter v =
    if v < n then raise Start_of_stream ;
    if v > m then raise End_of_stream ;
    let prev = counter in counter <- v ; prev
  method next () =
    self # set_counter (counter + 1) ;
  method reset () =
    self # set_counter n
end
class ['a] rewindable_between n m succ = object (self)
  inherit ['a] between n m succ
  method rewind =
    self # set_counter (counter - 1)
end
```

A method can de declared `virtual`:

- Its implementation is delayed to a child implementation.
- Syntax: `method virtual f : type`
- The class must be tagged virtual: `class virtual c args = body`.
- All inherited virtual methods must be implemented for the class to be concrete.
- Variables can also be virtual.
- Private methods can also be virtual.

Example: make stream a virtual generic implementation.

Starting from the original definition.

```
exception End_of_stream
exception Start_of_stream
class type ['a] stream = object
  method next : unit -> 'a
  method reset : unit -> unit
end
```

We write a generic builder, providing some virtual primitives.

```
class virtual ['a] generic_stream = object (self)
  val virtual mutable current : 'a
  method private virtual forward : unit -> unit
  method private virtual backward : unit -> unit
  method next () = self # forward () ; current
  method reset () =
    match self # backward () with
    | exception Start_of_stream -> ()
    | _ -> self#reset ()
end
```

And we derive a concrete implementation.

```
class ['a] between n m succ : [ 'a ] stream = object (self)
  inherit ['a] generic_stream
  val mutable current : 'a = n
  method private forward () =
    if current <= n then raise End_of_stream ;
    current <- current + 1
  method private backward () =
    if current >= m then raise Start_of_stream ;
    current <- current - 1
end
```

Patterns

Traits
Friend methods
Downcasting
Extensible mapper using objects

We can simulates the traits of other languages using virtual methods and multiple inheritance:

Example: rewindable, backtrackable streams.

```
class ['a] rewindable_backtrackable_between n m succ = object
  inherit ['a] between n m succ
  inherit ['a] backtrackable_stream
  inherit ['a] rewindable_stream
end
```

Traits 2 / 2

Where the traits express their minimum requirement on the main class.

```
class virtual ['a] rewindable_stream = object (self)
  val virtual mutable current : 'a
  method private virtual backward : unit -> unit
  method prev () = self # backward () ; current
end
class virtual ['a] backtrackable_stream = object
  val virtual mutable current : 'a
  val mutable stack = []
  method checkpoint () =
    stack <- current :: stack
  method backtrack () =
    match stack with
    | [] -> invalid_arg "backtrack"
    | v :: vs -> current <- v ; stack <- vs
end
```

No concept of friend methods in OCaml.

- Can be translated to methods callable only from the same module.
- Not directly available but can be simulated.
- Methods cannot be hidden from the outside, but they can be guarded.

Friend methods 2 / 2

Possible idea: requiring a friendship token.

```
module type STREAM = sig
  type friendship
  class stream : object
    method next : unit -> int
    method reset : friendship -> unit -> unit
  end
  val reset_stream : stream -> unit
end
module Stream : STREAM = struct
  type friendship = unit
  class stream = object
    method next () = 0
    method reset () () = ()
  end
  let reset_stream s = s#reset () ()
end
```

Other simpler or more complex encodings exists.

In closed contexts, downcasting can be simulated using a sum type.

```
type shape =
  < area : float ;
    upcast : shape ; downcast : classified_shape >
and circle =
  < area : float ; radius : float ;
    upcast : shape ; downcast : classified_shape >
and square =
  < area : float ; side : float ;
    upcast : shape ; downcast : classified_shape >
and classified_shape =
  Shape of shape| Circle of circle | Square of square
```

Implementation (needs a few coercions):

```
class make_shape area = object (self)
  method area = 3.14
  method upcast = (self :> shape)
  method downcast = Shape (self :> shape)
end
class make_square side = object (self)
  inherit make_shape (side *. side)
  method side = side
  method downcast = Square (self :> square)
end
class make_square radius = object (self)
  inherit make_shape (3.14 *. radius *. radius)
  method radius = radius
  method downcast = Circle (self :> circle)
end
```

Exercise: give a nicer solution with polymorphic variants.

Suppose we have the following mini language.

```
type expr =
  | Call of ident * expr list
  | Const of const
  | Var of ident
and instr =
  | Var_def of ident * expr
  | Fun_def of ident * ident list * instr list
  | If of expr * instr list * instr list
  | Expr of expr
and ident =
  string
and const =
  int
and program =
  instr list
```

We want to write rewriting passes over it.

In functional style, we start writing a `map` function that deep copies the structure.

```
let map p =
  let rec expr = function
    | Call (id, args) -> Call (ident id, List.map expr args)
    | Const c -> Const (const c)
    | Var id -> Var (ident id)
  and instr = function
    | Var_def (id, v) -> Var_def (ident id, expr v)
    | Fun_def (id, args, body) ->
      Fun_def (ident id, List.map ident args,
                    List.map instr body)
    | If (cond, bt, bf) ->
      If (expr cond, List.map instr bt, List.map instr bf)
    | Expr e -> Expr (expr e)
  and ident id = id and const c = c
  and program p = List.map instr p in
  program p
```

Then, we write rewriting functions by copy, paste and edit.

Extensible mapper using objects

This patterns translates easily to object style:

```
class map = object (self)
  method expr = function
    | Call (id, args) ->
      Call (self#ident id, List.map self#expr args)
    | Const c -> Const (self#const c)
    | Var id -> Var (self#ident id)
  method instr = function
    | Var_def (id, v) -> Var_def (self#ident id, self#expr v)
    | Fun_def (id, args, body) ->
      Fun_def (self#ident id, List.map self#ident args,
               List.map self#instr body)
    | If (cond, bt, bf) ->
      If (self#expr cond, List.map self#instr bt,
          List.map self#instr bf)
    | Expr e -> Expr (self#expr e)
  method ident id = id method const c = c
  method program p = List.map self#instr p
end
```

But instead of copy and paste, we can just use inheritance.

- Rewriting passes only contain their useful parts.
- The code is more readable and more resilient to type updates.
- Can be mixed with usual recursive descent.
- Less cumbersome than the functor based pattern.

Possible enhancements in real world cases:

- Use of exceptions to handle default cases.
- Use of exceptions to share sub-expressions.
- Functional or imperative folders.
- Higher order objects to combine multiple passes.

Example: uppercase all function names.

```
class uppercase_fun = object (self)
  inherit map as mom
  method! expr e =
    match mom#expr e with
    | Call (id, args) ->
      Call (String.uppercase id, args)
    | e -> e
  method! instr i =
    match mom#instr i with
    | Fun_def (id, args, body) ->
      Fun_def (String.uppercase id, args, body)
    | i -> i
end
```

Extensible mapper using objects

Example: precompute constant additions.

```
class precompute_additions = object (self)
  inherit map as mom
  method! expr e =
    match mom#expr e with
    | Call ("add", [ Const x ; Const y ]) -> Const (x + y)
    | e -> e
end
```

Outline