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, no overloading just overhiding
- 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 name_c [p_i p_j]

constraint typeexpr = typeexpr

val [**mutable**] ident = expr

initializer expr

method [**private**] [**virtual**] 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  a_i b_i =
  object
    val mutable a = a_i
    val mutable b = a_i
    method ga = a
    method mv (c , d) = a <- c; b<- d
    method ts () = Printf.sprintf "(%d, %d)" a b
end
```

- type p : `type p = < ga : 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#ga;;
- : int = 10
# p1#ts ();;
- : string = "( 10, 20)"
```

Inheritance from p

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

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

Late binding

```
class nc_p a_i b_i c_i =
  object
   inherits c_p a_i b_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 a_i b_i =
  object
    inherit printable () as super1
    inherit p a_i b_i 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 e = q <- q@[e]
  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 a_i b_i =
  object (self :'a)
    inherit p a_i b_i
    method eq (p : 'a) = self#ga == p#ga
  end
```

- type p_eq : μ'a.<eq : 'a -> bool; ga :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 a_i b_i =
  object
    inherit printable () as super1
    inherit p_eq a_i b_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 queue = <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