OCaml Objects and Classes

Mohammad Alam and Damith A. Karunaratne

CAS 706: Programming Languages

November 30, 2010

Outline

Classes and Objects

Classes can be defined using the **class** and **object** keywords.

$$
\begin{array}{l}
\textbf{\textit{class}}\ \textit{point} = \\
\qquad \textbf{\textit{object}} \\
\qquad\qquad \textbf{\textit{val mutable}}\ x = 0 \\
\qquad\qquad \textbf{\textit{method}}\ \textit{get\_x} = x \\
\qquad\qquad \textbf{\textit{method}}\ \textit{move}\ d = x \leftarrow x + d \\
\qquad \textbf{\textit{end;;}}
\end{array}
$$

$$
\begin{array}{l}
\textit{class point :} \\
\qquad \textit{object val mutable } x : \textit{int method get\_x} \\
\qquad : \textit{int method move} : \textit{int} \rightarrow \textit{unit end}
\end{array}
$$

- Class bodies are evaluated at creation time.

Initializing Classes

Classes can be initialized using the **new** keyword.

```
let x0 = ref 0;;

class point =
    object
        val mutable x = incr x0; !x0
        method get_x = x
        method move d = x <- x + d
    end;;

new point#get_x;;
- : int = 1

new point#get_x;;
- : int = 2
```

Initializing Classes with Parameters

Parameters can be specified after the class name.

```
class adjusted_point x_init =
    let origin = (x_init / 10) * 10 in
    object
        val mutable x = origin
        method get_x = x
        method get_offset = x - origin
        method move d = x <- x + d
    end;;
```

- Class bodies can contain expressions.

Referencing Self

If a reference to **self** is to be used, it must be explicitly bound.

- Binding will occur at invocation time.

```
let ints = ref [];;

class my_int =
    object (self)
        method n = 1
        method register = ints := self :: !ints
    end;;
```

If a reference to **self** is to be used, it must be explicitly bound.

- Binding will occur at invocation time.

```
let ints = ref [];;

class my_int =
    object (self)
        method n = 1
        method register = ints := self :: !ints
    end;;
```

Error: This expression has type < n : int; register : 'a; .. >
but an expression was expected of type 'b
Self type cannot escape its class

- Error caused by external an reference to **self**.

Initializers

Initializers can be used to evaluate an expression immediately after the object is built.

```
class adjusted_point x_init =
    let origin = (x_init / 10) * 10 in
    object (self)
        val mutable x = origin
        method get_x = x
        method get_offset = x - origin
        method move d = x <- x + d
        method print = print_int self#get_x
        initializer print_string "new point at ";
                    self#print; print_newline()
    end;;

let p = new printable_point 17;;
new point at 10
```

- Initializers cannot be overriden and are evaulated sequentially.

Immediate Objects

Objects can be created without a class.

$$\textbf{\textit{let}}\ minmax\ x\ y =$$
$$if\ x < y\ then\ \textbf{\textit{object method}}\ min = x\ \textbf{\textit{method}}\ max = y\ \textbf{\textit{end}}$$
$$else\ \textbf{\textit{object method}}\ min = y\ \textbf{\textit{method}}\ max = x\ \textbf{\textit{end}};;$$

$$val\ minmax : {}'a \rightarrow {}'a \rightarrow < max : {}'a; min : {}'a > = <fun>$$

- (+) Immediate objects can appear within expressions.
- (-) No type abbreviation.
- (-) Cannot inherit from immediate objects.

Virtual Methods

In OCaml, one can define **virtual methods** as well as **virtual instance variables**.

```
class virtual abstract_point =
    object
        val mutable virtual x : int
        method get_x = x
        method virtual move : int → unit
    end;;

class point x_init =
    object
        inherit abstract_point
        val mutable x = x_init
        method move d = x ← x + d
    end;;
```

- A class containing virtual methods or virtual instance variables must be flagged *virtual*, and cannot be *instantiated*.

Private Methods

A **private method** donot appear in the *object interface*.

$$
\begin{array}{l}
\textit{class restricted\_point x\_init} =\\
\qquad \textit{object (self)}\\
\qquad\qquad \textit{val mutable } x = x\_init\\
\qquad\qquad \textit{method get\_x} = x\\
\qquad\qquad \textit{method private move } d = x \leftarrow x + d\\
\qquad\qquad \textit{method bump} = self\#move\ 1\\
\qquad \textit{end;;}
\end{array}
$$

Type: $< get : int;\ bump : unit >$ *(in the context of type epression)*

- Private methods are inherited, i.e., they are visible in subclasses.
- Private methods can be made public in a subclass.
- Private methods can also be virtual and is defined like, *method private virtual identifer : type*.

Subtyping and Coercions

Subtype: A type t_1 is a subtype of t_2, written $t_1 <: t_2$, if values of type t_1 can be used where values of type t_2 are expected. Example,

$$type\ animal = < eat : unit >$$
$$type\ dog = < eat : unit; bark : unit >$$

Then subtyping relation dog $<:$ animal holds.

There are two forms of Subtyping:

1. **width subtyping:** It means if a subtype t_1 implements all the methods (and possibly more) of t_2 with the same method types.
2. **depth subtyping:** It is defined as: "If each method type s_i is a subtype of method type t_i, then the object type $< f_{1..n} : s_{1..n} >$ is a subtype of the object type $< f_{1..n} : t_{1..n} >$. [Hic08]"

Subtyping and Coercions (Cont..)

Coercions, in OCaml is never implicit or automatic. In a coercions ($obj : t_1 :> t2$), there are two necessary conditions:

1. the expression *obj* should have type t_1; and
2. type t_1 must be a subtype of t_2.

There are two ways to perform coercion:

1. Single coercion: ($object :> object\_type$)
2. Double coercion: ($object : object\_type :> object\_type$)

For example, if *colored_point* is a subtype of *points* then coercion can be done as following:

$$let\ colored\_point\_to\_points\ cp = (cp : colored\_point :> points);;$$

But, the following is not support in OCaml, as it is *narrowing coercion*.

$$(p : points :> colored\_point);;$$

Subtyping and Coercions (Cont..)

- The fully explicit coercion (Double coercion) is more precise and is sometimes unavoidable.
- Single coercion ($e :> t_2$) may fail if:
  - the type t_2 is recursive, or
  - the type t_2 has polymorphic structure

  The solution is to use fully explicit coercion ($e : t_1 :> t_2$).

Class Interface

Class interfaces are inferred from class definitions.

- Interfaces can be defined directly to restrict the type of a class.

$$
\begin{aligned}
&\textbf{\textit{class type}}\ restricted\_point\_type =\\
&\qquad \textbf{\textit{object}}\\
&\qquad\qquad \textbf{\textit{method}}\ get\_x : \textbf{\textit{int}}\\
&\qquad\qquad \textbf{\textit{method}}\ bump : \textbf{\textit{unit}}\\
&\qquad \textbf{\textit{end;;}}
\end{aligned}
$$

$$
\begin{aligned}
&class\ type\ restricted\_point\_type =\\
&\qquad object\ method\ bump : unit\ method\ get\_x : int\ end
\end{aligned}
$$

$$\textbf{\textit{fun}}\ (x : restricted\_point\_type) \rightarrow x;;$$

$$\text{-} : restricted\_point\_type \rightarrow restricted\_point\_type = <fun>$$

- Concrete instance variables and concrete private methods can be hidden.
- Public methods and and virtual members cannot be hidden.

Class Interface in Modules

Interfaces can be defined in module signatures in order to restrict the inferred signature of a module.

$$\begin{array}{l}
\textbf{\textit{module type}}\ POINT\ =\textbf{\textit{sig}}\\
\qquad \textbf{\textit{class}}\ restricted_point\ :\ \textbf{\textit{int}} \rightarrow\\
\qquad\qquad \textbf{\textit{object}}\\
\qquad\qquad\qquad \textbf{\textit{method}}\ get_x\ :\ \textbf{\textit{int}}\\
\qquad\qquad\qquad \textbf{\textit{method}}\ bump\ :\ \textbf{\textit{unit}}\\
\qquad\qquad \textbf{\textit{end}}\\
\textbf{\textit{end;;}}
\end{array}$$

$$\begin{array}{l}
module\ type\ POINT\ =\\
\qquad sig\\
\qquad\qquad class\ restricted_point\ :\\
\qquad\qquad\qquad int \rightarrow object\ method\ bump\ :\ unit\ method\\
\quad get_x\ :\ int\ end\\
end
\end{array}$$

Inheritance

Through inheritance, one may do the following:

- add new fields and new private methods
- add new public methods
- override fields or methods, but the type can't be changed

```
class animal species =
  object
    method eat = Printf.printf "A %s eats.\n" species
    method speak = Printf.printf "A %s speaks.\n" species
  end;;

class pet ~species ~owner ~name =
  object
    inherit animal species
    val owner = owner
    method name : string = name
    method eat =
      Printf.printf "A %s eats.\n" name;
      super#eat
  end;;
```

Inheritance Cont..

- **Subtyping** and **inheritance** are **not related**. Inheritance is a syntactic relation between classes while subtyping is a semantic relation between types. For example, in the previous example the class *'pet'* could have been defined directly, without inheriting from the class *'animal'*; the type of pet would remain unchanged and thus still be a subtype of animal.

Multiple Inheritance

- Inheritance could be done from multiple independent classes.
- Inheritance could be done from multiple virtual classes.
- Inheritance could be done from combination of independent and virtual classes. Virtual classes also inherits the same way.

```
class floatNumber =
  object
    inherit comparable
    inherit number
    ...
  end;;
```

- Previous definitions of a method can be reused by binding the related ancestor.
- The name super is a pseudo value identifier that can only be used to invoke a super-class method.

Multiple Inheritance Cont..

- Repeated inheritance is allowed in OCaml, i.e., a class can inherit another, along multiple paths (diamond problem) as well as directly. Example,

```
class a =
    object
        method x = 1
    end;;
class b =
    object
        inherit a
        method x = 2
        inherit a
    end;;
(new b)#x;;
```

Multiple Inheritance Cont..

- Example (cont..)

$$
\begin{array}{l}
\textit{class } a = \\
\quad \textit{object} \\
\quad\quad \textit{val mutable } x = 0 \\
\quad\quad \textit{method set } y = x \leftarrow y \\
\quad\quad \textit{method get} = x \\
\quad \textit{end;;} \\
\textit{class } b = \\
\quad \textit{object} \\
\quad\quad \textit{inherit } a \textit{ as } super1 \\
\quad\quad \textit{inherit } a \textit{ as } super2 \\
\quad\quad \textit{method test} = \\
\quad\quad\quad super1\#set\ 10; \\
\quad\quad\quad super2\#get \\
\quad \textit{end;;}
\end{array}
$$

In the above example, the mutable field x is duplicated not included textually.

Multiple Inheritance Cont..

- OCaml policy for **multiple inheritance**:
  1. *textual inclusion*, if methods are visible, then follows overriding rule: if a method is defined more than once, the last definition is used.
  2. *Duplication*, if fields and methods are hidden (private), then follows copy rule.
- Problem with textual inclusion is that, it might be necessary to know the text of all repeated superclasses.

Parameterized Classes

Reference cells can be implemented as objects but parameterized classes are needed (for non-immediate objects).

$$
\begin{array}{l}
\textbf{\textit{class}}\ ref\ x\_init\ = \\
\qquad \textbf{\textit{object}} \\
\qquad\qquad \textbf{\textit{val mutable}}\ x = x\_init \\
\qquad\qquad \textbf{\textit{method}}\ get = x \\
\qquad\qquad \textbf{\textit{method}}\ set\ y = x \leftarrow y \\
\qquad \textbf{\textit{end;;}}
\end{array}
$$

Parameterized Classes

Reference cells can be implemented as objects but parameterized classes are needed (for non-immediate objects).

```
class ref x_init =
    object
        val mutable x = x_init
        method get = x
        method set y = x <- y
    end;;
```

```
Error: Some type variables are unbound in this type:
    class ref :
      'a ->
      object
        val mutable x : 'a
        method get : 'a
        method set : 'a -> unit
      end
  The method get has type 'a where 'a is unbound
```

Class Type Parameters

Class type parameters are listed between **[** and **]**.

$$
\begin{array}{l}
\textbf{\textit{class}}\ [a]\ ref\ x\_init = \\
\qquad \textbf{\textit{object}} \\
\qquad\qquad \textbf{\textit{val mutable}}\ x = (x\_init : a) \\
\qquad\qquad \textbf{\textit{method}}\ get = x \\
\qquad\qquad \textbf{\textit{method}}\ set\ y = x \leftarrow y \\
\qquad \textbf{\textit{end;;}}
\end{array}
$$

$$
\begin{array}{l}
class\ [' a]\ ref : \\
\qquad 'a \rightarrow object\ val\ mutable\ x : 'a\ method\ get : 'a\ method\ set : \\
\qquad 'a \rightarrow unit\ end
\end{array}
$$

Constrained Class Type Parameters

The type parameters in the declaration can be constrained within the class definitions body.

```
class ['a] circle (c : 'a) =
    object
        constraint 'a = #point
            val mutable center = c
            method center = center
            method set_center c = center <- c
            method move = center#move
    end;;
```

Inheriting Parameterized Classes

Parameterized classes can be "specialized".

```
class ['a] colored_circle c =
    object
        constraint 'a = #colored_point
        inherit ['a] circle c
        method color = center#color
    end;;
```

- The type parameter must be specified when inheriting.
- Parameterized classes are polymorphic in their contents.

Polymorphic Methods

Parameterized classes on their own don't accommodate polymorphic methods.

```
class ['a] intlist (l : int list) =
    object
        method empty = (l = [])
        method fold f (accu : 'a) = List.fold_left f accu l
    end;;

let l = new intlist [1; 2; 3];;

    val l : '_a intlist = <obj>
```

Polymorphic Methods Cont..

l#fold (fun x y $\rightarrow$ *x+y) 0;;*

- : int = 6

l;;

- : int intlist = <*obj*>

*l#fold (**fun** s x* $\rightarrow$ *s ˆ string_of_int x ˆ " ") "";;*

Polymorphic Methods Cont..

l#fold (fun x y → x+y) 0;;

- : int = 6

l;;

- : int intlist = <obj>

*l#fold (**fun** s x → s ^ string_of_int x ^ " ") "";;*

Error: This expression has type int but an expression was expected of type string

- The objects are not polymorphic as the use of fold method fixes the type.

Polymorphic Methods Cont..

Instead of making the class polymorphic, make the method polymorphic.

$$
\begin{array}{l}
\textbf{\textit{class}}\ \mathit{intlist}\ (l : \mathit{int\ list}) = \\
\qquad \textbf{\textit{object}} \\
\qquad\qquad \textbf{\textit{method}}\ \mathit{empty} = (l = [\,]) \\
\qquad\qquad \textbf{\textit{method}}\ \mathit{fold} : {}'a.\ ({}'a \to \mathit{int} \to {}'a) \to {}'a \to {}'a = \\
\qquad\qquad\quad \mathit{fun}\ f\ \mathit{accu} \to \mathit{List.fold\_left}\ f\ \mathit{accu}\ l \\
\qquad \textbf{\textit{end}};;
\end{array}
$$

let *l* = ***new*** *intlist [1; 2; 3];;*

l#fold (fun x y → x+y) 0;;

- : int = 6

*l#fold (**fun** s x → s ^ string_of_int x ^ " ") "";;*

- : string = "1 2 3 "

Recursive Classes

- *Recursive classes* are used for objects whose types are mutually recursive. Example,

```
class window =
    object
        val mutable top_widget = (None : widget option)
        method top_widget = top_widget
    end
and widget (w : window) =
    object
        val window = w
        method window = window
    end;;
```

Here, the types are recursive but the classes are independent.

Functional Objects

Allows classes to have instance variables without assignments.

- Made possible through the {<...>} construct.

$$\begin{array}{l} \textbf{\textit{class}}\ functional\_point\ y = \\ \qquad \textbf{\textit{object}} \\ \qquad\qquad \textbf{\textit{val}}\ x = y \\ \qquad\qquad \textbf{\textit{method}}\ get\_x = x \\ \qquad\qquad \textbf{\textit{method}}\ move\ d = \{< x = x + d >\} \\ \qquad \textbf{\textit{end;;}} \end{array}$$

$$\begin{array}{l} class\ functional\_point : \\ \quad int \rightarrow \\ \quad object\ ('a)\ val\ x : int\ method\ get\_x : int\ method\ move : int \rightarrow \\ \quad 'a\ end \end{array}$$

$let\ p = new\ functional\_point\ 7;;$

$p\#get\_x;;$

$-\ :\ int = 7$

$(p\#move\ 3)\#get\_x;;$

$-\ :\ int = 10$

$p\#get\_x;;$

$-\ :\ int = 7$

- The **move** method is similar to a **binary method**.
  - A **binary method** takes an argument of the same type as **self**.
  - The **move** method takes a "copy" of the current **self** with some updates.

Object Cloning

Objects can be cloned using the **Oo.copy** library.

- Instance variables are copied, but their contents are shared.
  - eg. If the instance variable is a reference cell, the value will be shared.

$\boldsymbol{let}\ p = \boldsymbol{new}\ point\ 5;;$

$val\ p : point = <obj>$

$\boldsymbol{let}\ q = \boldsymbol{Oo.copy}\ p;;$

$val\ q : point = <obj>$

$p = q, p = p;;$

Object Cloning

Objects can be cloned using the **Oo.copy** library.

- Instance variables are copied, but their contents are shared.
  - eg. If the instance variable is a reference cell, the value will be shared.

```
let p = new point 5;;
val p : point = <obj>
let q = Oo.copy p;;
val q : point = <obj>
p = q, p = p;;
- : bool * bool = (false, true)
```

- Two objects are equal iff they are physically equal.

Cloning vs. Overriding

Cloning and overriding of objects function the same when used within objects.

$$
\begin{array}{l}
\boldsymbol{class}\ copy =\\
\quad\quad \boldsymbol{object}\\
\quad\quad\quad\quad \boldsymbol{method}\ copy = \{<\ >\}\\
\quad\quad \boldsymbol{end};;
\end{array}
$$

$$
\begin{array}{l}
\boldsymbol{class}\ copy =\\
\quad\quad \boldsymbol{object}\ (self)\\
\quad\quad\quad\quad \boldsymbol{method}\ copy = \boldsymbol{Oo.copy}\ self\\
\quad\quad \boldsymbol{end};;
\end{array}
$$

Friends

- In OCaml, the only way to share the representation between two different objects is to expose it to the whole world. For instance, using *binary methods*.
- A solution to this problem is to use the friends concept, where, all friends (classes or functions) defined within the same module share the same abstract view (signature) but knows the concrete representation. Thus, the concrete representation can be abstracted using signature. Example,

```
module type CURRENCY = sig
    type t
   class cur : float →
      object ('a)
        method v : t
        method plus : 'a → 'a
        method prod : float → 'a
      end
end;;
```

Friends Cont..

$$
\begin{array}{l}
\textit{module Currency} = \textit{struct} \\
\quad \textit{type } t = \textit{float} \\
\quad \textit{class cur } x = \\
\quad\quad \textit{object } (\_ : 'a) \\
\quad\quad\quad \textit{val } v = x \\
\quad\quad\quad \textit{method } v = v \\
\quad\quad\quad \textit{method plus}(z :' a) = \{< v = v +. z\#v >\} \\
\quad\quad\quad \textit{method prod } x = \{< v = x *. v >\} \\
\quad\quad \textit{end} \\
\textit{end;;}
\end{array}
$$

Module System vs Class System

OCaml has two very similar mechanism for modularity and abstraction: the **module system** and the **object system**.

- Before 3.12, the main difference was, one (object) is first class values and the other (module) is not. But, in 3.12 modules are first class values:
  - (**module** *module_expr* : *package_type*) converts the module (structure or functor) denoted by *module_expr* to a value that encapsulates the module.
  - (**val** *expr* : *package_type*) evaluates the expression *expr* to a value of type *package_type* (unpacking into a module).

For Example,

$$
\begin{aligned}
&\textit{module type DEVICE} = \textit{sig} \cdots \ \textit{end}\\
&\textit{module SVG} = \textit{struct} \cdots \ \textit{end}\\
&\textit{module PDF} = \textit{struct} \cdots \ \textit{end}\\
&\textit{let devices} : (\textit{string}, \textit{module DEVICE})\ \textit{Hashtbl.t} =\\
&\quad \textit{Hashtbl.create}\ 5\\
&\textit{let}\ \_ = \textit{Hashtbl.add devices}\ \text{"}\textit{SVG}\text{"}\ (\textit{module SVG} : \textit{DEVICE})\\
&\textit{let}\ \_ = \textit{Hashtbl.add devices}\ \text{"}\textit{PDF}\text{"}\ (\textit{module PDF} : \textit{DEVICE})\\
&\textit{module Device} = (\textit{val}\ (\textit{Hashtbl.find devices device\_name}) :\\
&\quad \textit{DEVICE})
\end{aligned}
$$

Module System vs Class System Cont..

- Modules can **contain type definitions** and objects cannot. This enables modules to provide privacy outside of the module boundaries. This advantage is used to implement the friends concept, as specified earlier.
- When dynamic scoping behavior is required, one would prefer object implementation rather than module.

```
class dog name =
  object (self)
    method name = name
    method eat = Printf.printf "%s eats.\n" self#name
    method bark = Printf.printf "%s barks!\n" self#name
    method bark_eat = self#bark; self#eat
  end;;
 class hound n =
  object (self)
    inherit dog n
    method bark = Printf.printf "%s howls!\n" self#name
  end;;
```

OOP of OCaml vs OOP of Java/C++

The relation between *object*, *class* and *type* in **Objective Caml** is very different from that in main stream object-oriented languages like **Java** or **C++**.

- In Java/C++ the **class name** is the **type of the object**; whereas, in OCaml object type is **set of public methods and their types**. Class types are abbreviated, it is the class name, but in the context of type expressions it stands for the object type.
- Types and classes in Objective Caml are **independent of each other**, i.e., **two unrelated classes** may produce objects of the **same type**, and there is no way at the type level to ensure that an object comes from a specific class.
- This also affects the **coercion**. In Java/C++, if a class hierarchy is defined as: animal $\leftarrow$ pet $\leftarrow$ petDog, then no other coersion than this hierarchy is allowed in Java/C++ but this is not the case in OCaml.
- In **Java/C++ subtyping and subclassing** are the **same** but not in OCaml.
- In Java/C++ object **cannot** be created without classes. In OCaml one can create **(immediate) objects** without going through the classes. These objects can be created inside an expression.

OOP of OCaml vs OOP of Java/C++ Cont..

- OCaml instance variables are private **cannot** be made **public**; in Java/C++ instance variables can be **defined as both**.
- In OCaml a subclass can make a **private superclass method, public**. In Java a subclass can only inherit a package-private member.
- In OCaml inheritance from **multiple independent class** is allowed, even the same superclass directly/indirectly. It is not the same for Java.
- Class type are similar to interface in Java. Java classes **can implement multiple interface** but OCaml classes can implement one type only. But OCaml type can inherit other type classes.
- In OCaml the class system is **statically typed** and in Java/C++ it is **dynamically typed**. That is why a pointer of a superclass can be used to refer to a subclass object dynamically, in Java/C++.
- In OCaml one can **initialize** as **instance variable** directly (without initializer) as well as indicate an **instance variable virtual**; **not** in Java/C++.

References

Jason Hickey.
Introduction to objective caml.
Preprint (January 11, 2008), available at
http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf, 2008.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon.
The Objective Caml System: Documentation and Users Manual.
INRIA, June 2010.
Available at
http://caml.inria.fr/pub/docs/manual-ocaml/index.html.

The caml language.
http://caml.inria.fr/, 2010.
[Online; Accessed: November 2, 2010].