15

Object-Oriented Programming

As you may have guessed from the name, Objective Caml supports object-oriented programming. Unlike imperative programming, in which execution is driven by explicit sequencing of operations, or functional programming, where it is driven by the required computations, object-oriented programming can be thought of as data driven. Using objects introduces a new organization of programs into classes of related objects. A class groups together data and operations. The latter, also known as *methods*, define the possible behaviors of an object. A method is invoked by *sending a message* to an object. When an object receives a message, it performs the action or the computation corresponding to the method specified by the message. This is different from applying a function to arguments because a message (which contains the method name) is sent to an object. It is up to the object itself to determine the code that will actually be executed; such a *delayed* binding between name and code makes behavior more adaptable and code easier to reuse.

With object-oriented programming, relations are defined between classes. Classes also define how objects communicate through message parameters. *Aggregation* and *inheritance* relations between classes allow new kinds of application modeling. A class that inherits from another class includes all definitions from the parent's class. However, it may extend the set of data and methods and redefine inherited behaviors, provided typing constraints are respected. We will use a graphical notation[1] to represent relations between classes.

Objective Caml's object extensions are integrated with the type system of the language: a class declaration defines a type with the same name as the class. Two kinds of polymorphism coexist. One of them is parametric polymorphism, which we have already seen with parameterized types: parameterized classes. The other one, known as *inclusion polymorphism*, uses the subtyping relation between objects and delayed binding. If the type of the class `sc` is a subtype of the class `c` then any object from `sc`

1. A number of notations exist for describing relations, e.g. UML (*Unified Modeling Language*).

may be used in place of an object from `c`. The subtype constraint must be stated explicitly. Inclusion polymorphism makes it possible to construct non-homogeneous lists where the type of each element is a subtype of a type common to all list elements. Since binding is delayed, sending the same message to all elements of such a list can activate different methods according to the sub-classes of the actual elements.

On the other hand, Objective Caml does not include the notion of method overloading, which would allow several definitions for one method name. Without this restriction, type inference might encounter ambiguous situations requiring additional information from the programmer.

It should be emphasized that Objective Caml is the only language with an object extension that provides both parameterized and inclusion polymorphism, while still being fully statically typed through type inference.

Chapter Plan

This chapter describes Objective Caml's object extension. This extension does not change any of the features of the language that we already studied in the previous chapters. A few new reserved keywords are added for the object-oriented syntax.

The first section describes class declaration syntax, object instantiation, and message passing. The second section explains the various relations that may exist between classes. The third section clarifies the notion of *object type* and demonstrates the richness of the object extension, thanks to abstract classes, multiple inheritance, and generic parameterized classes. The fourth section explains the subtyping relation and shows its power through inclusion polymorphism. The fifth section deals with a functional style of object-oriented programming, where the internal state of the object is not modified, but a modified copy of the receiving object is returned. The sixth section clarifies other parts of the object-oriented extension, such as interfaces and local declarations in classes, which allow class variables to be created.

Classes, Objects, and Methods

The object-oriented extension of Objective Caml is integrated with the functional and imperative kernels of the language, as well as with its type system. Indeed, this last point is unique to the language. Thus we have an object-oriented, statically typed language, with type inference. This extension allows definition of classes and instances, class inheritance (including multiple inheritance), parameterized classes, and abstract classes. Class interfaces are generated from their definition, but may be made more precise through a signature, similarly to what is done for modules.

Object-Oriented Terminology

We summarize below the main object-oriented programming terms.

class: a *class* describes the contents of the objects that belong to it: it describes an aggregate of data fields (called *instance variables*), and defines the operations (called *methods*).

object: an object is an element (or *instance*) of a class; objects have the behaviors of their class. The object is the actual component of programs, while the class specifies how instances are created and how they behave.

method: a method is an action which an object is able to perform.

sending a message *sending a message* to an object means asking the object to execute or *invoke* one of its methods.

Class Declaration

The simplest syntax for defining a class is as follows. We shall develop this definition throughout this chapter.

Syntax :

```
class name p1 ... pn =
  object
      ⋮
    instance variables
      ⋮
    methods
      ⋮
  end
```

$p_1, \ldots, p_n$ are the parameters for the constructor of the class; they are omitted if the class has no parameters.

An instance variable is declared as follows:

Syntax :

```
val name = expr
or
val mutable name = expr
```

When a data field is declared **mutable**, its value may be modified. Otherwise, the value is always the one that was computed when *expr* was evaluated during object creation.

Methods are declared as follows:

Syntax :

```
method name p1 ... pn = expr
```

Other clauses than **val** and **method** can be used in a class declaration: we shall introduce them as needed.

Our first class example. We start with the unavoidable class `point`:

- the data fields `x` and `y` contain the coordinates of the point,
- two methods provide access to the data fields (`get_x` and `get_y`),
- two displacement methods (`moveto`: absolute displacement) and (`rmoveto`: relative displacement),
- one method presents the data as a *string* (`to_string`),
- one method computes the distance to the point from the origin (`distance`).

```
# class point (x_init,y_init) =
    object
      val mutable x = x_init
      val mutable y = y_init
      method get_x = x
      method get_y = y
      method moveto (a,b) =  x <- a ; y <- b
      method rmoveto (dx,dy) =  x <- x + dx ; y <- y + dy
      method to_string () =
          "( " ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
      method distance () = sqrt (float(x*x + y*y))
    end ;;
```

Note that some methods do not need parameters; this is the case for `get_x` and `get_y`. We usually access instance variables with parameterless methods.

After we declare the class `point`, the system prints the following text:

```
class point :
  int * int ->
  object
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

This text contains two pieces of information. First, the type for objects of the class; this type will be abbreviated as *point*. The type of an object is the list of names and types of methods in its class. In our example, *point* is an abbreviation for:

```
  < distance : unit → unit; get_x : int; get_y : int;
    moveto : int * int → unit; rmoveto : int * int → unit;
    to_string : unit → unit >
```

Next, we have a constructor for instances of class `point`, whose type is *int*int* `-->` *point*. The constructor allows us to construct `point` objects (we´ll just say "`point`s" to be brief) from the initial values provided as arguments. In this case, we construct a

`point` from a pair of integers (meaning the initial position). The constructor `point` is used with the keyword **`new`**.

It is possible to define class types:

```
# type simple_point = < get_x : int; get_y : int; to_string : unit → unit > ;;
type simple_point = < get_x : int; get_y : int; to_string : unit -> unit >
```

Note

Type *`point`* does not repeat all the informations shown after a class declaration. Instance variables are not shown in the type. Only methods have access to these instance variables.

Warning A class declaration is a type declaration. As a consequence, it cannot contain a free type variable.

We will come back to this point later when we deal with type constraints (page 454) and parameterized classes (page 460).

A Graphical Notation for Classes

We adapt the UML notation for the syntax of Objective Caml types. Classes are denoted by a rectangle with three parts:

- the top part shows the name of the class,
- the middle part lists the attributes (data fields) of a class instance,
- the bottom part shows the methods of an instance of the class.

Figure 15.1 gives an example of the graphical representation for the class `caml`.

Figure 15.1: Graphical representation of a class.

Type information for the fields and methods of a class may be added.

Instance Creation

An object is a value of a class, called an *instance* of the class. Instances are created with the generic construction primitive `new`, which takes the class and initialization values as arguments.

Syntax : **new** *name* $expr_1 \ldots expr_n$

The following example creates several instances of class ***point***, from various initial values.

```
# let p1 = new point (0,0);;
val p1 : point = <obj>
# let p2 = new point (3,4);;
val p2 : point = <obj>
# let coord = (3,0);;
val coord : int * int = 3, 0
# let p3 = new point coord;;
val p3 : point = <obj>
```

In Objective Caml, the constructor of a class is unique, but you may define your own specific function `make_point` for point creation:

```
# let make_point x = new point (x,x) ;;
val make_point : int -> point = <fun>
# make_point 1 ;;
- : point = <obj>
```

Sending a Message

The notation `#` is used to send a message to an object. [2]

Syntax : obj_1**#***name* $p_1 \ldots p_n$

The message with method name "*name*" is sent to the object *obj*. The arguments p_1, $\ldots$, p_n are as expected by the method *name*. The method must be defined by the class of the object, i.e. visible in the type. The types of arguments must conform to the types of the formal parameters. The following example shows several queries performed on objects from the class `point`.

```
# p1#get_x;;
- : int = 0
# p2#get_y;;
- : int = 4
# p1#to_string();;
- : string = "( 0, 0)"
# p2#to_string();;
```

2. In most object-oriented languages, a dot notation is used. However, the dot notation was already used for records and modules, so a new symbol was needed.

```
- : string = "( 3, 4)"
# if (p1#distance()) = (p2#distance())
   then print_string ("That's just chance\n")
   else print_string ("We could bet on it\n");;
We could bet on it
- : unit = ()
```

From the type point of view, objects of type ***point*** can be used by polymorphic functions of Objective Caml, just as any other value in the language:

```
# p1 = p1 ;;
- : bool = true
# p1 = p2;;
- : bool = false
# let l = p1::[];;
val l : point list = [<obj>]
# List.hd l;;
- : point = <obj>
```

Warning Object equality is defined as physical equality.

We shall clarify this point when we study the subtyping relation (page 469).

Relations between Classes

Classes can be related in two ways:

1. An aggregation relation, named *Has-a*:
   class C_2 is related by *Has-a* with class C_1 when C_2 has a field whose type is that of class C_1. This relation can be generalized as: C_2 has at least one field whose type is that of class C_1.
2. An inheritance relation, named *Is-a*:
   class C_2 is a subclass of class C_1 when C_2 extends the behavior of C_1. One big advantage of object-oriented programming is the ability to extend the behavior of an existing class while reusing the code written for the original class. When a class is extended, the new class inherits all the fields (data and methods) of the class being extended.

Aggregation

Class C_1 aggregates class C_2 when at least one of its instance variables has type C_2. One gives the arity of the aggregation relation when it is known.

An Example of Aggregation

Let us define a figure as a set of points. Therefore we declare class *picture* (see figure 15.2), in which one of the fields is an array of points. Then the class *picture* aggregates *point*, using the generalized relation *Has-a*.

```
# class picture n =
    object
      val mutable ind = 0
      val tab = Array.create n (new point(0,0))
      method add p =
        try  tab.(ind)<-p ; ind <- ind + 1
        with Invalid_argument("Array.set")
                → failwith ("picture.add:ind =" ^ (string_of_int ind))
      method remove () = if (ind > 0) then ind <-ind-1
      method to_string () =
        let s = ref "["
        in for i=0 to ind-1 do s:= !s ^ " " ^ tab.(i)#to_string() done ;
           (!s) ^ "]"
    end ;;
class picture :
  int ->
  object
    val mutable ind : int
    val tab : point array
    method add : point -> unit
    method remove : unit -> unit
    method to_string : unit -> string
  end
```

To build a figure, we create an instance of class *picture*, and insert the points as required.

```
# let pic = new picture 8;;
val pic : picture = <obj>
# pic#add p1; pic#add p2; pic#add p3;;
- : unit = ()
# pic#to_string ();;
- : string = "[ ( 0, 0) ( 3, 4) ( 3, 0)]"
```

A Graphical Notation for Aggregation

The relation between class *picture* and class *point* is represented graphically in figure 15.2. An arrow with a diamond at the tail represents aggregation. In this example, class *picture* has 0 or more points. Furthermore, we show above the arrow the arity of the relation.

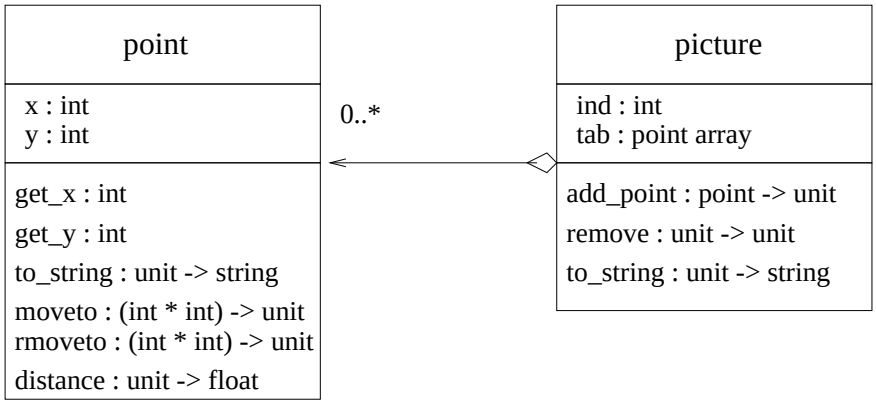

Figure 15.2: Aggregation relation.

Inheritance Relation

This is the main relation in object-oriented programming. When class `c2` inherits from class `c1`, it inherits all fields from the parent class. It can also define new fields, or redefine inherited methods to specialize them. Since the parent class has not been modified, the applications using it do not need to be adapted to the changes introduced in the new class.

The syntax of inheritance is as follows:

Syntax : **`inherit`** $name_1$ p_1 $\ldots$ p_n [**`as`** $name_2$]

Parameters p_1, $\ldots$, p_n are what is expected from the constructor of class $name_1$. The optional keyword **`as`** associates a name with the parent class to provide access to its methods. This feature is particularly useful when the child class redefines a method of the parent class (see page 445).

An Example of Simple Inheritance

Using the classic example, we can extend class ***`point`*** by adding a color attribute to the points. We define the class ***`colored_point`*** inheriting from class ***`point`***. The color is represented by the field `c` of type ***`string`***. We add a method `get_color` that returns the value of the field. Finally, the string conversion method is overridden to recognize the new attribute.

Note

The `x` and `y` variables seen in `to_string` are the fields, not the class initialization arguments.

```
# class colored_point (x,y) c =
    object
```

```
      inherit point (x,y)
      val mutable c = c
      method get_color = c
      method set_color nc = c <- nc
      method to_string () = "( " ^ (string_of_int x) ^
                            ", " ^ (string_of_int y) ^ ")" ^
                            " [" ^ c ^ "] "
    end ;;
class colored_point :
  int * int ->
  string ->
  object
    val mutable c : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_color : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method set_color : string -> unit
    method to_string : unit -> string
  end
```

The constructor arguments for *colored_point* are the pair of coordinates required for the construction of a *point* and the color for the colored point.

The methods inherited, newly defined or redefined correspond to the behaviors of instances of the class.

```
# let pc = new colored_point (2,3) "white";;
val pc : colored_point = <obj>
# pc#get_color;;
- : string = "white"
# pc#get_x;;
- : int = 2
# pc#to_string();;
- : string = "( 2, 3) [white] "
# pc#distance;;
- : unit -> float = <fun>
```

We say that the class *point* is a *parent* class of class *colored_point* and that the latter is the *child* of the former.

Warning When redefining a method in a child class, you must respect the method type defined in the parent class.

A Graphical Notation for Inheritance

The inheritance relation between classes is denoted by an arrow from the child class to the parent class. The head of the arrow is a closed triangle. In the graphical represen-

tation of inheritance, we only show the new fields and methods, and redefined methods in the child class. Figure 15.3 displays the relation between class *colored_point* and its parent *point*.

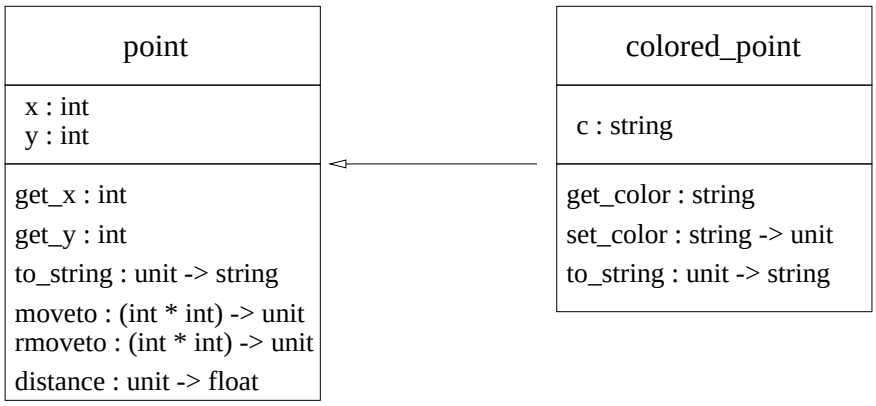

Figure 15.3: Inheritance Relation.

Since it contains additional methods, type *colored_point* differs from type *point*. Testing for equality between instances of these classes produces a long error message containing the whole type of each class, in order to display the differences.

```
# p1 = pc;;
Characters 6-8:
This expression has type
  colored_point =
    < distance : unit -> float; get_color : string; get_x : int; get_y :
      int; moveto : int * int -> unit; rmoveto : int * int -> unit;
      set_color : string -> unit; to_string : unit -> string >
but is here used with type
  point =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string >
Only the first object type has a method get_color
```

Other Object-oriented Features

References: `self` *and* `super`

When defining a method in a class, it may be convenient to be able to invoke a method from a parent class. For this purpose, Objective Caml allows the object itself, as well as (the objects of) the parent class to be named. In the former case, the chosen name is given after the keyword **`object`**, and in the latter, after the inheritance declaration.

For example, in order to define the method `to_string` of colored points, it is better to invoke the method `to_string` from the parent class and to extend its behavior with a new method, `get_color`.

```
# class colored_point (x,y) c  =
    object (self)
      inherit point (x,y) as super
      val c = c
      method get_color = c
      method to_string () = super#to_string() ^  " [" ^ self#get_color ^ "] "
    end ;;
```

Arbitrary names may be given to the parent and child class objects, but the names `self` and `this` for the current class and `super` for the parent are conventional. Choosing other names may be useful with multiple inheritance since it makes it easy to differentiate the parents (see page 457).

Warning You may not reference a variable of an instance's parent if you declare a new variable with the same name since it masks the former.

Delayed Binding

With *delayed binding* the method used when a message is sent is decided at runtime; this is opposed to static binding where the decision is made at compile time. In Objective Caml, delayed binding of methods is used; therefore, the exact piece of code to be executed is determined by the recipient of the message.

The above declaration of class *colored_point* redefines the method `to_string`. This new definition uses method `get_color` from this class. Now let us define another class *colored_point_1*, inheriting from *colored_point*; this new class redefines method `get_color` (testing that the character string is appropriate), but does not redefine `to_string`.

```
# class colored_point_1 coord c =
    object
      inherit colored_point coord  c
      val true_colors = ["white"; "black"; "red"; "green"; "blue"; "yellow"]
      method get_color = if List.mem c true_colors then c else "UNKNOWN"
    end ;;
```

Method `to_string` is the same in both classes of colored points; but two objects from these classes will have a different behavior.

```
# let p1 = new colored_point (1,1) "blue as an orange" ;;
val p1 : colored_point = <obj>
# p1#to_string();;
- : string = "( 1, 1) [blue as an orange] "
# let p2 = new colored_point_1 (1,1) "blue as an orange" ;;
```

```
val p2 : colored_point_1 = <obj>
# p2#to_string();;
- : string = "( 1, 1) [UNKNOWN] "
```

The binding of `get_color` within `to_string` is not fixed when the class *`colored_point`* is compiled. The code to be executed when invoking the method `get_color` is determined from the methods associated with instances of classes *`colored_point`* and *`colored_point_1`*. For an instance of *`colored_point`*, sending the message `to_string` causes the execution of `get_color`, defined in class *`colored_point`*. On the other hand, sending the same message to an instance of *`colored_point_1`* invokes the method from the parent class, and the latter triggers method `get_color` from the child class, controlling the relevance of the string representing the color.

Object Representation and Message Dispatch

An object is split in two parts: one may vary, the other is fixed. The varying part contains the instance variables, just as for a record. The fixed part corresponds to a methods table, shared by all instances of the class.

The methods table is a sparse array of functions. Every method name in an application is given a unique id that serves as an index into the methods table. We assume the existence of a machine instruction `GETMETHOD(o,n)`, that takes two parameters: an object `o` and an index `n`. It returns the function associated with this index in the methods table. We write `f_n` for the result of the call `GETMETHOD(o,n)`. Compiling the message send `o#m` computes the index `n` of the method name `m` and produces the code for applying `GETMETHOD(o,n)` to object `o`. This corresponds to applying function `f_n` to the receiving object `o`. Delayed binding is implemented through a call to `GETMETHOD` at run time.

Sending a message to `self` within a method is also compiled as a search for the index of the message, followed by a call to the function found in the methods table.

In the case of inheritance, since the method name always has the same index, regardless of redefinition, only the entry in new class' methods table is changed for redefinitions. So sending message `to_string` to an instance of class `point` will apply the conversion function of a point, while sending the same message to an instance of `colored_point` will find at the same index the function corresponding to the method which has been redefined to recognize the color field.

Thanks to this index invariance, subtyping (see page 465) is insured to be coherent with respect to the execution. Indeed if a colored point is explicitly constrained to be a point, then upon sending the message `to_string` the method index from class `point` is computed, which coincides with that from class `colored_point`. Searching for the method will be done within the table associated with the receiving instance, i.e. the `colored_point` table.

Although the actual implementation in Objective Caml is different, the principle of dynamic search for the method to be used is still the same.

Initialization

The class definition keyword **initializer** is used to specify code to be executed during object construction. An initializer can perform any computation and field access that is legal in a method.

Syntax : **initializer** *expr*

Let us again extend the class `point`, this time by defining a verbose point that will announce its creation.

```
# class verbose_point p =
    object(self)
      inherit point p
      initializer
        let xm = string_of_int x and ym = string_of_int y
        in Printf.printf ">> Creation of a point at (%s %s)\n"
                         xm ym ;
           Printf.printf "   , at distance %f from the origin\n"
                         (self#distance()) ;
    end ;;

# new verbose_point (1,1);;
>> Creation of a point at (1 1)
   , at distance 1.414214 from the origin
- : verbose_point = <obj>
```

An amusing but instructive use of initializers is tracing class inheritance on instance creation. Here is an example:

```
# class c1 =
    object
      initializer print_string "Creating an instance of c1\n"
    end ;;

# class c2 =
    object
      inherit c1
      initializer print_string "Creating an instance of c2\n"
    end ;;

# new c1 ;;
Creating an instance of c1
- : c1 = <obj>
# new c2 ;;
Creating an instance of c1
Creating an instance of c2
- : c2 = <obj>
```

Constructing an instance of *c2* requires first constructing an instance of the parent class.

Private Methods

A method may be declared *private* with the keyword **private**. It will appear in the interface to the class but not in instances of the class. A private method can only be invoked from other methods; it cannot be sent to an instance of the class. However, private methods are inherited, and therefore can be used in definitions of the hierarchy[3].

Syntax : **method private** *name* = *expr*

Let us extend the class *point*: we add a method `undo` that revokes the last move. To do this, we must remember the position held before performing a move, so we introduce two new fields, `old_x` and `old_y`, together with their update method. Since we do not want the user to have direct access to this method, we declare it as private. We redefine the methods `moveto` and `rmoveto`, keeping note of the current position before calling the previous methods for performing a move.

```
# class point_m1 (x0,y0) =
    object(self)
      inherit point (x0,y0) as super
      val mutable old_x = x0
      val mutable old_y = y0
      method private mem_pos () = old_x <- x ; old_y <- y
      method undo () = x <- old_x; y <- old_y
      method moveto (x1, y1) = self#mem_pos () ; super#moveto (x1, y1)
      method rmoveto (dx, dy) = self#mem_pos () ; super#rmoveto (dx, dy)
  end ;;
class point_m1 :
  int * int ->
  object
    val mutable old_x : int
    val mutable old_y : int
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method private mem_pos : unit -> unit
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
    method undo : unit -> unit
  end
```

We note that method `mem_pos` is preceded by the keyword **private** in type *point_m1*. It can be invoked from within method `undo`, but not on another instance. The situation is the same as for instance variables. Even though fields `old_x` and `old_y` appear in the results shown by compilation, that does not imply that they may be handled directly (see page 438).

```
# let p = new point_m1 (0, 0) ;;
```

3. The **private** of Objective Caml corresponds to **protected** of Objective C, C++ and Java

```
val p : point_m1 = <obj>
# p#mem_pos() ;;
Characters 0-1:
This expression has type point_m1
It has no method mem_pos
# p#moveto(1, 1) ; p#to_string() ;;
- : string = "( 1, 1)"
# p#undo() ; p#to_string() ;;
- : string = "( 0, 0)"
```

Warning A type constraint may make public a method declared with attribute **private**.

Types and Genericity

Besides the ability to model a problem using aggregation and inheritance relations, object-oriented programming is interesting because it provides the ability to reuse or modify the behavior of classes. However, extending an Objective Caml class must preserve the static typing properties of the language.

With abstract classes, you can factorize code and group their subclasses into one "communication protocol". An abstract class fixes the names and types of messages that may be received by instances of child classes. This technique will be better appreciated in connection with multiple inheritance.

The notion of an *open object type* (or simply an *open type*) that specifies the required methods allows code to work with instances using generic functions. But you may need to make the type constraints precise; this will be necessary for parameterized classes, which provide the genericity of parameterized polymorphism in the context of classes. With this latter object layer feature, Objective Caml can really be generic.

Abstract Classes and Methods

In abstract classes, some methods are declared without a body. Such methods are called *abstract*. It is illegal to instantiate an abstract class; **new** cannot be used. The keyword **virtual** is used to indicate that a class or method is abstract.

Syntax : **class virtual** *name* **= object** ... **end**

A class *must* be declared abstract as soon as one of its methods is abstract. A method is declared abstract by providing only the method type.

Syntax : **method virtual** *name* : *type*

When a subclass of an abstract class redefines *all* of the abstract methods of its parent, then it may become concrete; otherwise it also has to be declared abstract.

As an example, suppose we want to construct a set of displayable objects, all with a method `print` that will display the object's contents translated into a character string. All such objects need a method `to_string`. We define class `printable`. The string may vary according to the nature of the objects that we consider; therefore method `to_string` is abstract in the declaration of `printable` and consequently the class is also abstract.

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

We note that the abstractness of the class and of its method `to_string` is made clear in the type we obtain.

From this class, let us try to define the class hierarchy of figure 15.4.

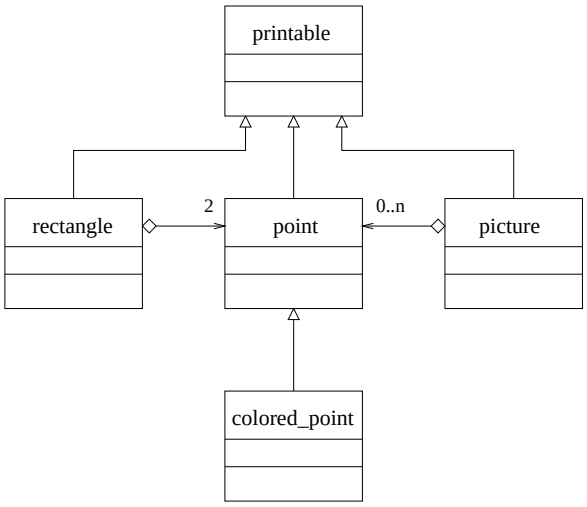

Figure 15.4: Relations between classes of displayable objects.

It is easy to redefine the classes *point*, *colored_point* and *picture* by adding to their declarations a line `inherit printable ()` that provides them with a method *print* through inheritance.

```
# let p = new point (1,1) in p#print() ;;
( 1, 1)- : unit = ()
# let pc = new colored_point (2,2) "blue" in pc#print() ;;
( 2, 2) with color blue- : unit = ()
# let t = new picture 3 in t#add (new point (1,1)) ;
```

```
                          t#add (new point (3,2)) ;
                          t#add (new point (1,4)) ;
                          t#print() ;;
[ ( 1, 1) ( 3, 2) ( 1, 4)]- : unit = ()
```

Subclass *rectangle* below inherits from *printable*, and defines method `to_string`. Instance variables `llc` (resp. `urc`) mean the lower left corner point (resp. upper right corner point) in the rectangle.

```
# class rectangle (p1,p2) =
    object
      inherit printable ()
      val llc = (p1 : point)
      val urc = (p2 : point)
      method to_string () = "[" ^ llc#to_string() ^ "," ^ urc#to_string() ^ "]"
    end ;;
class rectangle :
  point * point ->
  object
    val llc : point
    val urc : point
    method print : unit -> unit
    method to_string : unit -> string
  end
```

Class *rectangle* inherits from the abstract class *printable*, and thus receives method `print`. It has two instance variables of type *point*: the lower left corner (`llc`) and upper right corner. Method `to_string` sends the message `to_string` to its `point` instance variables `llc` and `urc`.

```
# let r = new rectangle (new point (2,3), new point (4,5));;
val r : rectangle = <obj>
# r#print();;
[( 2, 3),( 4, 5)]- : unit = ()
```

Classes, Types, and Objects

You may remember that the type of an object is determined by the type of its methods. For instance, the type *point*, inferred during the declaration of class *point*, is an abbreviation for type:

```
point =
  < distance : unit -> float; get_x : int; get_y : int;
    moveto : int * int -> unit; rmoveto : int * int -> unit;
    to_string : unit -> string  >
```

This is a closed type; that is, all methods and associated types are fixed. No additional methods and types are allowed. Upon a class declaration, the mechanism of type inference computes the closed type associated with class.

Open Types

Since sending a message to an object is part of the language, you may define a function that sends a message to an object whose type is still undefined.

```
# let f x = x#get_x ;;
val f : < get_x : 'a; .. > -> 'a = <fun>
```

The type inferred for the argument of `f` is an object type, since a message is sent to `x`, but this object type is *open*. In function `f`, parameter `x` must have at least a method `get_x`. Since the result of sending this message is not used within function `f`, its type has to be as general as possible (i.e. a variable of type *'a*). So type inference allows the function `f` to be used with any object having a method `get_x`. The double points (..) at the end of the type *< get_x : 'a; .. >* indicate that the type of `x` is open.

```
# f (new point(2,3)) ;;
- : int = 2
# f (new colored_point(2,3) "emerald") ;;
- : int = 2
# class c () =
    object
      method get_x = "I have a method get_x"
    end ;;
class c : unit -> object method get_x : string end
# f (new c ()) ;;
- : string = "I have a method get_x"
```

Type inference for classes may generate open types, particularly for initial values in instance construction. The following example constructs a class *couple*, whose initial values `a` and `b` have a method `to_string`.

```
# class couple (a,b) =
    object
      val p0 = a
      val p1 = b
      method to_string() = p0#to_string() ^ p1#to_string()
      method copy () = new couple (p0,p1)
    end ;;
class couple :
  (< to_string : unit -> string; .. > as 'a) *
  (< to_string : unit -> string; .. > as 'b) ->
  object
    val p0 : 'a
    val p1 : 'b
    method copy : unit -> couple
    method to_string : unit -> string
```

```
  end
```

The types of both `a` and `b` are open types, with method `to_string`. We note that these two types are considered to be different. They are marked "*as 'a*" and "*as 'b*", respectively. Variables of types *'a* and *'b* are constrained by the generated type.

We use the sharp symbol to indicate the open type built from a closed type *obj_type*:

Syntax : *#obj_type*

The type obtained contains all of the methods of type *obj_type* and terminates with a double point.

Type Constraints.

In the chapter on functional programming (see page 28), we showed how an expression can be constrained to have a type more precise than what is produced by inference. Object types (open or closed) can be used to enhance such constraints. One may want to open *a priori* the type of a defined object, in order to apply it to a forthcoming method. We can use an open object constraint:

Syntax : **(***name***:#***type***)**

Which allows us to write:

```
# let g (x : #point) = x#message;;
val g :
  < distance : unit -> float; get_x : int; get_y : int; message : 'a;
    moveto : int * int -> unit; print : unit -> unit;
    rmoveto : int * int -> unit; to_string : unit -> string; .. > ->
  'a = <fun>
```

The type constraint with *#point* forces `x` to have at least all of the methods of *point*, and sending message "`message`" adds a method to the type of parameter `x`.

Just as in the rest of the language, the object extension of Objective Caml provides static typing through inference. When this mechanism does not have enough information to determine the type of an expression, a type variable is assigned. We have just seen that this process is also valid for typing objects; however, it sometimes leads to ambiguous situations which the user must resolve by explicitly giving type information.

```
# class a_point p0 =
    object
      val p = p0
      method to_string() = p#to_string()
    end ;;
Characters 6-89:
Some type variables are unbound in this type:
  class a_point :
    (< to_string : unit -> 'b; .. > as 'a) ->
    object val p : 'a method to_string : unit -> 'b end
The method to_string has type unit -> 'a where 'a is unbound
```

We resolve this ambiguity by saying that parameter p0 has type ***#point***.

```
# class a_point (p0 : #point) =
    object
      val p = p0
      method to_string() = p#to_string()
    end ;;
class a_point :
  (#point as 'a) -> object val p : 'a method to_string : unit -> string end
```

In order to set type constraints in several places in a class declaration, the following syntax is used:

Syntax : **constraint** $type_1 = type_2$

The above example can be written specifying that parameter p0 has type ***'a***, then putting a type constraint upon variable ***'a***.

```
# class a_point (p0 : 'a) =
    object
      constraint 'a = #point
      val p = p0
      method to_string() = p#to_string()
    end ;;
class a_point :
  (#point as 'a) -> object val p : 'a method to_string : unit -> string end
```

Several type constraints can be given in a class declaration.

Warning An open type cannot appear as the type of a method.

This strong restriction exists because an open type contains an uninstantiated type variable coming from the rest of the type. Since one cannot have a free variable type in a type declaration, a method containing such a type is rejected by type inference.

```
# class b_point p0  =
    object
      inherit a_point p0
      method get = p
    end ;;
Characters 6-77:
Some type variables are unbound in this type:
  class b_point :
    (#point as 'a) ->
    object val p : 'a method get : 'a method to_string : unit -> string end
The method get has type #point where .. is unbound
```

In fact, due to the constraint "**constraint** *'a = #point*", the type of get is the open type ***#point***. The latter contains a free variable type noted by a double point (..), which is not allowed.

Inheritance and the Type of `self`

There exists an exception to the prohibition of a type variable in the type of methods: a variable may stand for the type of the object itself (`self`). Consider a method testing the equality between two points.

```
# class point_eq (x,y) =
    object (self : 'a)
      inherit point (x,y)
      method eq (p:'a) = (self#get_x = p#get_x) && (self#get_y = p#get_y)
    end ;;
class point_eq :
  int * int ->
  object ('a)
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method eq : 'a -> bool
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

The type of method `eq` is *`'a -> bool`*, but the type variable stands for the type of the instance at construction time.

You can inherit from the class *`point_eq`* and redefine the method `eq`, whose type is still parameterized by the instance type.

```
# class colored_point_eq (xc,yc) c =
    object (self : 'a)
      inherit point_eq (xc,yc) as super
      val c = (c:string)
      method get_c = c
      method eq (pc : 'a) =  (self#get_x = pc#get_x) && (self#get_y = pc#get_y)
                          && (self#get_c = pc#get_c)
    end ;;
class colored_point_eq :
  int * int ->
  string ->
  object ('a)
    val c : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method eq : 'a -> bool
    method get_c : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
```

```
    method to_string : unit -> string
  end
```

The method `eq` from class *colored_point_eq* still has type *'a -> bool*; but now the variable *'a* stands for the type of an instance of class *colored_point_eq*. The definition of `eq` in class *colored_point_eq* masks the inherited one. Methods containing the type of the instance in their type are called binary methods. They will cause some limitations in the subtyping relation described in page 465.

Multiple Inheritance

With multiple inheritance, you can inherit data fields and methods from several classes. When there are identical names for fields or methods, only the last declaration is kept, according to the order of inheritance declarations. Nevertheless, it is possible to reference a method of one of the parent classes by associating different names with the inherited classes. This is not true for instance variables: if an inherited class masks an instance variable of a previously inherited class, the latter is no longer directly accessible. The various inherited classes do not need to have an inheritance relation. Multiple inheritance is of interest because it increases class reuse.

Let us define the abstract class *geometric_object*, which declares two virtual methods `compute_area` and `compute_peri` for computing the area and perimeter.

```
# class virtual geometric_object () =
  object
    method virtual compute_area : unit → float
    method virtual compute_peri : unit → float
  end;;
```

Then we redefine class `rectangle` as follows:

```
# class rectangle_1 ((p1,p2) :'a) =
  object
    constraint 'a = point * point
    inherit printable ()
    inherit geometric_object ()
    val llc = p1
    val urc = p2
    method to_string () =
     "["^llc#to_string()^","^urc#to_string()^"]"
    method compute_area() =
     float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
    method compute_peri() =
     float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
  end;;
class rectangle_1 :
  point * point ->
  object
    val llc : point
    val urc : point
```

```
    method compute_area : unit -> float
    method compute_peri : unit -> float
    method print : unit -> unit
    method to_string : unit -> string
  end
```

This implementation of classes respects the inheritance graph of figure 15.5.

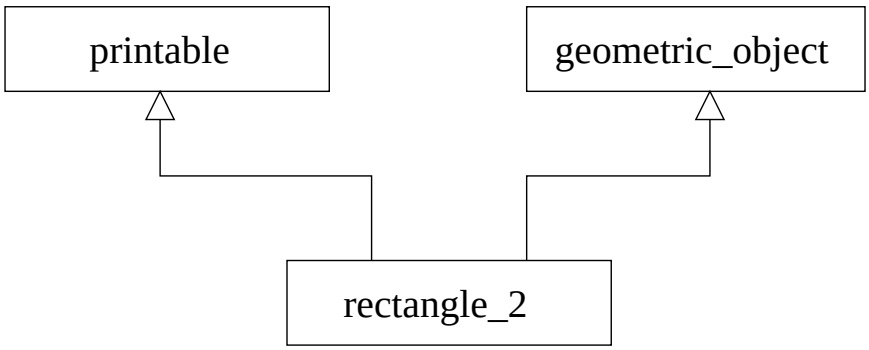

Figure 15.5: Multiple inheritance.

In order to avoid rewriting the methods of class `rectangle`, we may directly inherit from `rectangle`, as shown in figure 15.6.

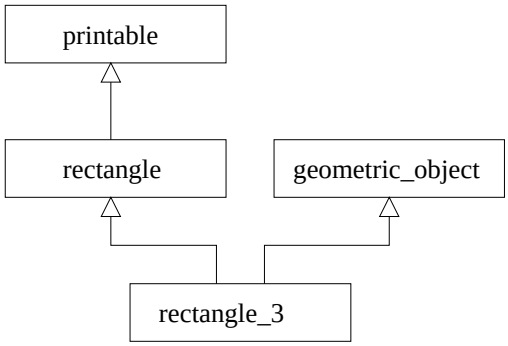

Figure 15.6: Multiple inheritance (continued).

In such a case, only the abstract methods of the abstract class `geometric_object` must be defined in `rectangle_2`.

```
# class rectangle_2 (p2 :'a) =
    object
    constraint 'a = point * point
    inherit rectangle p2
    inherit geometric_object ()
    method compute_area() =
```

```
    float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
  method compute_peri() =
    float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
end;;
```

Continuing in the same vein, the hierarchies `printable` and `geometric_object` could have been defined separately, until it became useful to have a class with both behaviors. Figure 15.7 displays the relations defined in this way.

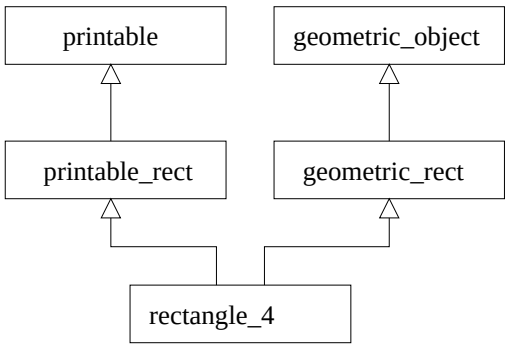

Figure 15.7: Multiple inheritance (end).

If we assume that classes `printable_rect` and `geometric_rect` define instance variables for the corners of a rectangle, we get class `rectangle_3` with four points (two per corner).

```
class rectangle_3 (p1,p2) =
 inherit printable_rect (p1,p2) as super_print
 inherit geometric_rect (p1,p2) as super_geo
end;;
```

In the case where methods of the same type exist in both classes `..._rect`, then only the last one is visible. However, by naming parent classes (`super_...`), it is always possible to invoke a method from either parent.

Multiple inheritance allows factoring of the code by integrating methods already written from various parent classes to build new entities. The price paid is the size of constructed objects, which are bigger than necessary due to duplicated fields, or inherited fields useless for a given application. Furthermore, when there is duplication (as in our last example), communication between these fields must be established manually (update, etc.). In the last example for class `rectangle_3`, we obtain instance variables of classes `printable_rect` and `geometric_rect`. If one of these classes has a method which modifies these variables (such as a scaling factor), then it is necessary to propagate these modifications to variables inherited from the other class. Such a heavy communication between inherited instance variables often signals a poor modeling of the given problem.

Parameterized Classes

Parameterized classes let Objective Caml's parameterized polymorphism be used in classes. As with the type declarations of Objective Caml, class declarations can be parameterized with type variables. This provides new opportunities for genericity and code reuse. Parameterized classes are integrated with ML-like typing when type inference produces parameterized types.

The syntax differs slightly from the declaration of parameterized types; type parameters are between brackets.

Syntax : **class [***'a, 'b, ...***]** *name* **= object** ... **end**

The Objective Caml type is noted as usual: *('a,'b,...) name*.

For instance, if a class `pair` is required, a naive solution would be to set:

```
# class pair x0 y0 =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ;;
Characters 6-106:
Some type variables are unbound in this type:
  class pair :
    'a ->
    'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
The method fst has type 'a where 'a is unbound
```

One again gets the typing error mentioned when class *a_point* was defined (page 452). The error message says that type variable *'a*, assigned to parameter `x0` (and therefore to `x` and `fst`), is not bound.

As in the case of parameterized types, it is necessary to parameterize class *pair* with two type variables, and force the type of construction parameters `x0` and `y0` to obtain a correct typing:

```
# class ['a,'b] pair (x0:'a) (y0:'b) =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ;;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

Type inference displays a class interface parameterized by variables of type `'a` and `'b`.

When a value of a parameterized class is constructed, type parameters are instantiated with the types of the construction parameters:

```
# let p = new pair 2 'X';;
val p : (int, char) pair = <obj>
# p#fst;;
- : int = 2
# let q = new pair 3.12 true;;
val q : (float, bool) pair = <obj>
# q#snd;;
- : bool = true
```

Note

In class declarations, type parameters are shown between brackets, but in types, they are shown between parentheses.

Inheritance of Parameterized Classes

When inheriting from a parameterized class, one has to indicate the parameters of the class. Let us define a class *acc_pair* that inherits from *('a,'b) pair*; we add two methods for accessing the fields, `get1` and `get2`,

```
# class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
    object
      inherit ['a,'b] pair x0 y0
      method get1 z = if x = z then y else raise Not_found
      method get2 z = if y = z then x else raise Not_found
    end;;
class ['a, 'b] acc_pair :
  'a ->
  'b ->
  object
    val x : 'a
    val y : 'b
    method fst : 'a
    method get1 : 'a -> 'b
    method get2 : 'b -> 'a
    method snd : 'b
  end
# let p = new acc_pair 3 true;;
val p : (int, bool) acc_pair = <obj>
# p#get1 3;;
- : bool = true
```

We can make the type parameters of the inherited parameterized class more precise, e.g. for a pair of points.

```
# class point_pair (p1,p2) =
    object
      inherit [point,point] pair p1 p2
    end;;
class point_pair :
  point * point ->
  object
```

```
    val x : point
    val y : point
    method fst : point
    method snd : point
  end
```

Class *point_pair* no longer needs type parameters, since parameters *'a* and *'b* are completely determined.

To build pairs of displayable objects (i.e. having a method `print`), we reuse the abstract class `printable` (see page 451), then we define the class `printable_pair` which inherits from `pair`.

```
# class printable_pair x0 y0 =
    object
      inherit [printable, printable] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end;;
```

This implementation allows us to construct pairs of instances of `printable`, but it cannot be used for objects of another class with a method `print`.

We could try to open type `printable` used as a type parameter for `acc_pair`:

```
# class printable_pair (x0 ) (y0 ) =
    object
      inherit [ #printable, #printable ] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end;;
Characters 6-149:
Some type variables are unbound in this type:
  class printable_pair :
    (#printable as 'a) ->
    (#printable as 'b) ->
    object
      val x : 'a
      val y : 'b
      method fst : 'a
      method get1 : 'a -> 'b
      method get2 : 'b -> 'a
      method print : unit -> unit
      method snd : 'b
    end
The method fst has type #printable where .. is unbound
```

This first attempt fails because methods `fst` and `snd` contain an open type.

So we shall keep the type parameters of the class, while constraining them to the open type *#printable*.

```
# class ['a,'b] printable_pair (x0 ) (y0 ) =
    object
      constraint 'a = #printable
      constraint 'b = #printable
```

```
      inherit ['a,'b] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end;;
class ['a, 'b] printable_pair :
  'a ->
  'b ->
  object
    constraint 'a = #printable
    constraint 'b = #printable
    val x : 'a
    val y : 'b
    method fst : 'a
    method get1 : 'a -> 'b
    method get2 : 'b -> 'a
    method print : unit -> unit
    method snd : 'b
  end
```

Then we construct a displayable pair containing a point and a colored point.

```
# let pp = new printable_pair
             (new point (1,2)) (new colored_point (3,4) "green");;
val pp : (point, colored_point) printable_pair = <obj>
# pp#print();;
( 1, 2)( 3, 4) with color green- : unit = ()
```

Parameterized Classes and Typing

From the point of view of types, a parameterized class is a parameterized type. A value of such a type can contain weak type variables.

```
# let r = new pair [] [];;
val r : ('_a list, '_b list) pair = <obj>
# r#fst;;
- : '_a list = []
# r#fst = [1;2];;
- : bool = false
# r;;
- : (int list, '_a list) pair = <obj>
```

A parameterized class can also be viewed as a closed object type; therefore nothing prevents us from also using it as an open type with the sharp notation.

```
# let compare_nothing ( x : ('a, 'a) #pair) =
    if x#fst = x#fst then x#mess else x#mess2;;
val compare_nothing :
  < fst : 'a; mess : 'b; mess2 : 'b; snd : 'a; .. > -> 'b = <fun>
```

This lets us construct parameterized types that contain weak type variables that are also open object types.

```
# let prettytype x ( y : ('a, 'a) #pair) = if x = y#fst then y else y;;
val prettytype : 'a -> (('a, 'a) #pair as 'b) -> 'b = <fun>
```

If this function is applied to one parameter, we get a closure, whose type variables are weak. An open type, such as ***#pair***, still contains uninstantiated parts, represented by the double point (..). In this respect, an open type is a partially known type parameter. Upon weakening such a type after a partial application, the displayer specifies that the type variable representing this open type has been weakened. Then the notation is *_#pair*.

```
# let g = prettytype 3;;
val g : ((int, int) _#pair as 'a) -> 'a = <fun>
```

Now, if function g is applied to a pair, its weak type is modified.

```
# g (new acc_pair 2 3);;
- : (int, int) acc_pair = <obj>
# g;;
- : (int, int) acc_pair -> (int, int) acc_pair = <fun>
```

Then we can no longer use g on simple pairs.

```
# g (new pair 1 1);;
Characters 4-16:
This expression has type (int, int) pair = < fst : int; snd : int >
but is here used with type
  (int, int) acc_pair =
    < fst : int; get1 : int -> int; get2 : int -> int; snd : int >
Only the second object type has a method get1
```

Finally, since parameters of the parameterized class can also get weakened, we obtain the following example.

```
# let h = prettytype [];;
val h : (('_b list, '_b list) _#pair as 'a) -> 'a = <fun>
# let h2 = h (new pair [] [1;2]);;
val h2 : (int list, int list) pair = <obj>
# h;;
- : (int list, int list) pair -> (int list, int list) pair = <fun>
```

The type of the parameter of h is no longer open. The following application cannot be typed because the argument is not of type ***pair***.

```
# h (new acc_pair [] [4;5]);;
Characters 4-25:
This expression has type
  ('a list, int list) acc_pair =
    < fst : 'a list; get1 : 'a list -> int list; get2 : int list -> 'a list;
      snd : int list >
but is here used with type
```

```
  (int list, int list) pair = < fst : int list; snd : int list >
Only the first object type has a method get1
```

Note

Parameterized classes of Objective Caml are absolutely necessary as soon as one has methods whose type includes a type variable different from the type of `self`.

Subtyping and Inclusion Polymorphism

Subtyping makes it possible for an object of some type to be considered and used as an object of another type. An object type *ot2* could be a subtype of *ot1* if:

1. it includes all of the methods of *ot1*,
2. each method of *ot2* that is a method of *ot1* is a subtype of the *ot1* method.

The subtype relation is only meaningful between objects: it can only be expressed between objects. Furthermore, the subtype relation must always be explicit. It is possible to indicate either that a type is a subtype of another, or that an object has to be considered as an object of a super type.

Syntax :

```
(name : sub_type :> super_type)
(name :> super_type)
```

Example

Thus we can indicate that an instance of `colored_point` can be used as an instance of `point`:

```
# let pc = new colored_point (4,5) "white";;
val pc : colored_point = <obj>
# let p1 = (pc : colored_point :> point);;
val p1 : point = <obj>
# let p2 = (pc :> point);;
val p2 : point = <obj>
```

Although known as a point, `p1` is nevertheless a colored point, and sending the method `to_string` will trigger the method relevant for colored points:

```
# p1#to_string();;
- : string = "( 4, 5) with color white"
```

This way, it is possible to build lists containing both points and colored points:

```
# let l = [new point (1,2) ; p1] ;;
val l : point list = [<obj>; <obj>]
# List.iter (fun x → x#print(); print_newline()) l;;
( 1, 2)
```

```
( 4, 5) with color white
- : unit = ()
```

Of course, the actions that can be performed on the objects of such a list are restricted to those allowed for points.

```
# p1#get_color () ;;
Characters 1-3:
This expression has type point
It has no method get_color
```

The combination of delayed binding and subtyping provides a new form of polymorphism: *inclusion polymorphism*. This is the ability to handle values of any type having a subtype relation with the expected type. Although static typing information guarantees that sending a message will always find the corresponding method, the behavior of the method depends on the actual receiving object.

Subtyping is not Inheritance

Unlike mainstream object-oriented languages such as C++, Java, and SmallTalk, subtyping and inheritance are different concepts in Objective Caml. There are two main reasons for this.

1. Instances of the class `c2` may have a type that is a subtype of the object type `c1` even if the class `c2` does not inherit from the class `c1`. Indeed, the class `colored_point` can be defined independently from the class `point`, provided the type of its instances are constrained to the object type `point`.
2. Class `c2` may inherit from the class `c1` but have instances whose type is not a subtype of the object type `c1`. This is illustrated in the following example, which uses the ability to define an abstract method that takes an as yet undetermined instance as an argument of the class being defined. In our example, this is method `eq` of class `equal`.

```
# class virtual equal () =
    object(self:'a)
     method virtual eq : 'a → bool
    end;;
class virtual equal : unit -> object ('a) method virtual eq : 'a -> bool end
# class c1 (x0:int) =
    object(self)
     inherit equal ()
     val x = x0
     method get_x = x
     method eq o = (self#get_x = o#get_x)
    end;;
class c1 :
  int ->
  object ('a) val x : int method eq : 'a -> bool method get_x : int end
```

```
# class c2 (x0:int) (y0:int) =
    object(self)
     inherit equal ()
     inherit c1 x0
     val y = y0
     method get_y = y
     method eq o = (self#get_x = o#get_x) && (self#get_y = o#get_y)
    end;;
class c2 :
  int ->
  int ->
  object ('a)
    val x : int
    val y : int
    method eq : 'a -> bool
    method get_x : int
    method get_y : int
  end
```

We cannot force the type of an instance of `c2` to be the type of instances of `c1`:

```
# let a = ((new c2 0 0) :> c1) ;;
Characters 11-21:
This expression cannot be coerced to type
  c1 = < eq : c1 -> bool; get_x : int >;
it has type c2 = < eq : c2 -> bool; get_x : int; get_y : int >
but is here used with type < eq : c1 -> bool; get_x : int; get_y : int >
Type c2 = < eq : c2 -> bool; get_x : int; get_y : int >
is not compatible with type c1 = < eq : c1 -> bool; get_x : int >
Only the first object type has a method get_y
```

Types *c1* and *c2* are incompatible because the type of `eq` in `c2` is not a subtype of the type of `eq` in `c1`. To see why this is true, let `o1` be an instance of *c1*. If `o21` were an instance of *c2* subtyped to *c1*, then since `o21` and `o1` would both be of type *c1* the type of `eq` in *c2* would be a subtype of the type of `eq` in *c1* and the expression `o21#eq(o1)` would be correctly typed. But at run-time, since `o21` is an instance of class *c2*, the method `eq` of *c2* would be triggered. But this method would try to send the message `get_y` to `o1`, which does not have such a method; our type system would have failed!

For our type system to fulfill its role, the subtyping relation must be defined less naïvely. We do this in the next paragraph.

Formalization

Subtyping between objects. 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 $i \in \{1, \ldots, n\}$.

Function call. If $f : t \to s$, and if $a : t'$ and $t' \leq t$ then (fa) is well typed, and has type s.

Intuitively, a function f expecting an argument of type t may safely receive 'an argument of a subtype t' of t.

Subtyping of functional types. Type $t' \to s'$ is a subtype of $t \to s$, denoted by $t' \to s' \leq t \to s$, 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*. Although surprising at first, this relation between functional types can easily be justified in the context of object-oriented programs with dynamic binding.

Let us assume that two classes $c1$ and $c2$ both have a method m. Method m has type $t_1 \to s_1$ in $c1$, and type $t_2 \to s_2$ in $c2$. For the sake of readability, let us denote by $m_{(1)}$ the method m of $c1$ and $m_{(2)}$ that of $c2$. Finally, let us assume $c2 \leq c1$, i.e. $t_2 \to s_2 \leq t_1 \to s_1$, and let us look at a simple example of the covariance and contravariance relations.

Let $g : s_1 \to \alpha$, and $h\ (o : c1)\ (x : t_1)\ =\ g(o\#m(x))$

Covariance: function h expects an object of type $c1$ as its first argument. Since $c2 \leq c1$, it is legal to pass it an object of type $c2$. Then the method invoked by $o\#m(x)$ is $m_{(2)}$, which returns a value of type s_2. Since this value is passed to g which expects an argument of type s_1, clearly we must have $s_2 \leq s_1$.

Contravariance: for its second argument, h requires a value of type t_1. If, as above, we give h a first argument of type $c2$, then method $m_{(2)}$ is invoked. Since it expects an argument of type t_2, $t_1 \leq t_2$.

Inclusion Polymorphism

By "polymorphism" we mean the ability to apply a function to arguments of any "shape" (type), or to send a message to objects of various shapes.

In the context of the functional/imperative kernel of the language, we have already seen parameterized polymorphism, which enables you to apply a function to arguments of arbitrary type. The polymorphic parameters of the function have types containing type variables. A polymorphic function will execute the same code for various types of parameters. To this end, it will not depend on the structure of these arguments.

The subtyping relation, used in conjunction with delayed binding, introduces a new kind of polymorphism for methods: inclusion polymorphism. It lets the same message be sent to instances of different types, provided they have been constrained to the same subtype. Let us construct a list of points where some of them are in fact colored points treated as points. Sending the same message to all of them triggers the execution

of different methods, depending on the class of the receiving instance. This is called inclusion polymorphism because it allows messages from class `c`, to be sent to any instance of class `sc` that is a subtype of `c` (*`sc :> c`*) that has been constrained to `c`. Thus we obtain a polymorphic message passing for all classes of the tree of subtypes of `c`. Contrary to parameterized polymorphism, the code which is executed may be different for these instances.

Thanks to parameterized classes, both forms of polymorphism can be used together.

Equality between Objects

Now we can explain the somewhat surprising behavior of structural equality between objects which was presented on page 441. Two objects are structurally equal when they are physically the same.

```
# let p1 =  new point (1,2);;
val p1 : point = <obj>
# p1  = new point (1,2);;
- : bool = false
# p1 = p1;;
- : bool = true
```

This comes from the subtyping relation. Indeed we can try to compare an instance `o2` of a class *`sc`* that is a subtype of `c`, constrained to `c`, with an instance of `o1` from class `c`. If the fields which are common to these two instances are equal, then these objects might be considered as equal. This is wrong from a structural point of view because `o2` could have additional fields. Therefore Objective Caml considers that two objects are structurally different when they are physically different.

```
# let pc1 = new colored_point (1,2) "red";;
val pc1 : colored_point = <obj>
# let q = (pc1 :> point);;
val q : point = <obj>
# p1 = q;;
- : bool = false
```

This restrictive view of equality guarantees that an answer `true` is not wrong, but an answer `false` guarantees nothing.

Functional Style

Object-oriented programming usually has an imperative style. A message is sent to an object that physically modifies its internal state (i.e. its data fields). It is also possible to use a functional approach to object-oriented programming: sending a message returns a new object.

Object Copy

Objective Caml provides a special syntactic construct for returning a copy of an object `self` with some of the fields modified.

Syntax : {**<** *name*$_1$**=***expr*$_1$**;**...; *name*$_n$**=***expr*$_n$ **>**}

This way we can define functional points where methods for relative moves have no side effect, but instead return a new point.

```
# class f_point p =
    object
     inherit point p
     method f_rmoveto_x (dx) = {< x = x + dx >}
     method f_rmoveto_y (dy) = {< y = y + dy >}
    end ;;
class f_point :
  int * int ->
  object ('a)
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method f_rmoveto_x : int -> 'a
    method f_rmoveto_y : int -> 'a
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

With the new methods, movement no longer modifies the receiving object; instead a new object is returned that reflects the movement.

```
# let p = new f_point (1,1) ;;
val p : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# let q = p#f_rmoveto_x 2 ;;
val q : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# print_string (q#to_string()) ;;
( 3, 1)- : unit = ()
```

Since these methods construct an object, it is possible to send a message directly to the result of the method `f_rmoveto_x`.

```
# print_string ((p#f_rmoveto_x 3)#to_string()) ;;
( 4, 1)- : unit = ()
```

The result type of the methods `f_rmoveto_x` and `f_rmoveto_y` is the type of the instance of the defined class, as shown by the *'a* in the type of `f_rmoveto_x`.

```
# class f_colored_point (xc, yc) (c:string) =
    object
      inherit f_point(xc, yc)
      val color = c
      method get_c = color
    end ;;
class f_colored_point :
  int * int ->
  string ->
  object ('a)
    val color : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method f_rmoveto_x : int -> 'a
    method f_rmoveto_y : int -> 'a
    method get_c : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

Sending `f_rmoveto_x` to an instance of `f_colored_point` returns a new instance of `f_colored_point`.

```
# let fpc = new f_colored_point (2,3) "blue" ;;
val fpc : f_colored_point = <obj>
# let fpc2 = fpc#f_rmoveto_x 4 ;;
val fpc2 : f_colored_point = <obj>
# fpc2#get_c;;
- : string = "blue"
```

One can also obtain a copy of an arbitrary object, using the the primitive `copy` from module `Oo`:

```
# Oo.copy ;;
- : (< .. > as 'a) -> 'a = <fun>
# let q = Oo.copy p ;;
val q : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# print_string (q#to_string()) ;;
( 1, 1)- : unit = ()
# p#moveto(4,5) ;;
- : unit = ()
# print_string (p#to_string()) ;;
( 4, 5)- : unit = ()
```

```
# print_string (q#to_string()) ;;
( 1, 1)- : unit = ()
```

Example: a Class for Lists

A functional method may use the object itself, `self`, to compute the value to be returned. Let us illustrate this point by defining a simple hierarchy of classes for representing lists of integers.

First we define the abstract class, parameterized by the type of list elements.

```
# class virtual ['a] o_list () =
  object
    method virtual empty : unit → bool
    method virtual cons  : 'a → 'a o_list
    method virtual head :  'a
    method virtual tail : 'a o_list
  end;;
```

We define the class of non empty lists.

```
# class  ['a] o_cons (n ,l)  =
  object (self)
    inherit ['a] o_list ()
    val car = n
    val cdr = l
    method empty () = false
    method cons x = new o_cons (x, (self : 'a #o_list :> 'a o_list))
    method head = car
    method tail = cdr
  end;;
class ['a] o_cons :
  'a * 'a o_list ->
  object
    val car : 'a
    val cdr : 'a o_list
    method cons : 'a -> 'a o_list
    method empty : unit -> bool
    method head : 'a
    method tail : 'a o_list
  end
```

We should note that method `cons` returns a new instance of *`'a o_cons`*. To this effect, the type of `self` is constrained to *`'a #o_list`*, then subtyped to *`'a o_list`*. Without subtyping, we would obtain an open type (*`'a #o_list`*), which appears in the type of the methods, and is strictly forbidden (see page 456). Without the additional constraint, the type of `self` could not be a subtype of *`'a o_list`*.

This way we obtain the expected type for method `cons`. So now we know the trick and we define the class of empty lists.

```
# exception EmptyList ;;
# class ['a] o_nil () =
    object(self)
     inherit ['a] o_list ()
     method empty () = true
     method cons x = new o_cons (x, (self : 'a #o_list :> 'a o_list))
     method head = raise EmptyList
     method tail = raise EmptyList
    end ;;
```

First of all we build an instance of the empty list, and then a list of integers.

```
# let i = new o_nil ();;
val i : '_a o_nil = <obj>
# let l = new o_cons (3,i);;
val l : int o_list = <obj>
# l#head;;
- : int = 3
# l#tail#empty();;
- : bool = true
```

The last expression sends the message `tail` to the list containing the integer 3, which triggers the method `tail` from the class *'a o_cons*. The message `empty()`, which returns `true`, is sent to this result. You can see that the method which has been executed is `empty` from the class *'a o_nil*.

Other Aspects of the Object Extension

In this section we describe the declaration of "object" types and local declarations in classes. The latter can be used for class variables by making constructors that reference the local environment.

Interfaces

Class interfaces are generally infered by the type system, but they can also be defined by a type declaration. Only public methods appear in this type.

Syntax :

```
class type name =
  object
    ⋮
    val name_i : type_i
    ⋮
    method name_j : type_j
    ⋮
  end
```

Thus we can define the class `point` interface:

```
# class type interf_point  =
  object
    method get_x : int
    method get_y : int
    method moveto : (int * int ) → unit
    method rmoveto : (int * int )  → unit
    method to_string : unit → string
    method distance : unit → float
  end ;;
```

This declaration is useful because the defined type can be used as a type constraint.

```
# let seg_length (p1:interf_point) (p2:interf_point) =
   let x = float_of_int (p2#get_x - p1#get_x)
   and y = float_of_int (p2#get_y - p1#get_y) in
    sqrt ((x*.x) +. (y*.y)) ;;
val seg_length : interf_point -> interf_point -> float = <fun>
```

Interfaces can only mask fields of instance variables and private methods. They cannot mask abstract or public methods.

This is a restriction in their use, as shown by the following example:

```
# let p = ( new point_m1 (2,3) : interf_point);;
Characters 11-29:
This expression has type
  point_m1 =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string; undo : unit -> unit >
but is here used with type
  interf_point =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string >
Only the first object type has a method undo
```

Nevertheless, interfaces may use inheritance. Interfaces are especially useful in combination with modules: it is possible to build the signature of a module using object types, while only making available the description of class interfaces.

Local Declarations in Classes

A class declaration produces a type and a constructor. In order to make this chapter easier to read, we have been presenting constructors as functions without an environment. In fact, it is possible to define constructors which do not need initial values to create an instance: that means that they are no longer functional. Furthermore one

can use local declarations in the class. Local variables captured by the constructor are shared and can be treated as class variables.

Constant Constructors

A class declaration does not need to use initial values passed to the constructor. For example, in the following class:

```
# class example1 =
    object
      method print () = ()
    end ;;
class example1 : object method print : unit -> unit end
# let p = new example1 ;;
val p : example1 = <obj>
```

The instance constructor is constant. The allocation does not require an initial value for the instance variables. As a rule, it is better to use an initial value such as `()`, in order to preserve the functional nature of the constructor.

Local Declarations for Constructors

A local declaration can be written directly with abstraction.

```
# class example2 =
    fun a →
      object
        val mutable r = a
        method get_r = r
        method plus x = r <- r + x
     end;;
class example2 :
  int ->
  object val mutable r : int method get_r : int method plus : int -> unit end
```

Here it is easier to see the functional nature of the constructor. The constructor is a closure which may have an environment that binds free variables to an environment of declarations. The syntax for class declarations allows local declarations in this functional expression.

Class Variables

Class variables are declarations which are known at class level and therefore shared by all instances of the class. Usually these class variables can be used outside of any instance creation. In Objective Caml, thanks to the functional nature of a constructor with a non-empty environment, we can make these values (particularly the modifiable ones) shared by all instances of a class.

We illustrate this facility with the following example, which allows us to keep a register of the number of instances of a class. To do this we define a parameterized abstract class *'a om*.

```
# class virtual  ['a] om =
    object
      method  finalize ()  = ()
      method virtual destroy : unit → unit
      method virtual to_string : unit → string
      method virtual all : 'a list
    end;;
```

Then we declare class `'a lo`, whose constructor contains local declarations for `n`, which associates a unique number with each instance, and for `l`, which contains the list of pairs (number, instance) of still active instances.

```
# class ['a] lo =
    let l = ref []
    and n = ref 0 in
      fun s →
        object(self: 'b )
          inherit ['a] om
          val mutable num = 0
          val name = s
          method to_string () = s
          method print () = print_string s
          method print_all () =
            List.iter (function (a,b) →
                         Printf.printf "(%d,%s) " a (b#to_string())) !l
          method destroy () = self#finalize();
                              l:= List.filter (function (a,b) → a <> num) !l; ()
          method all = List.map snd !l
          initializer incr n; num <- !n;  l:= (num, (self :> 'a  om) ) :: !l ; ()
        end;;
class ['a] lo :
  string ->
  object
    constraint 'a = 'a om
    val name : string
    val mutable num : int
    method all : 'a list
    method destroy : unit -> unit
    method finalize : unit -> unit
    method print : unit -> unit
    method print_all : unit -> unit
    method to_string : unit -> string
  end
```

At each creation of an instance of class *lo*, the initializer increments the reference `n` and adds the pair (number, `self`) to the list `l`. Methods `print` and `print_all` display respectively the receiving instance and all the instances containing in `l`.

```
# let m1 = new lo "start";;
val m1 : ('a om as 'a) lo = <obj>
```

```
# let m2 = new lo "between";;
val m2 : ('a om as 'a) lo = <obj>
# let m3 = new lo "end";;
val m3 : ('a om as 'a) lo = <obj>
# m2#print_all();;
(3,end) (2,between) (1,start) - : unit = ()
# m2#all;;
- : ('a om as 'a) list = [<obj>; <obj>; <obj>]
```

Method `destroy` removes an instance from the list of instances, and calls method `finalize` to perform a last action on this instance before it disappears from the list. Method `all` returns all the instances of a class created with **`new`**.

```
# m2#destroy();;
- : unit = ()
# m1#print_all();;
(3,end) (1,start) - : unit = ()
# m3#all;;
- : ('a om as 'a) list = [<obj>; <obj>]
```

We should note that instances of subclasses are also kept in this list. Nothing prevents you from using the same technique by specializing some of these subclasses. On the other hand, the instances obtained by a copy (`Oo.copy` or {**`< >`**}) are not tracked.

Exercises

Stacks as Objects

Let us reconsider the stacks example, this time in object oriented style.

1. Define a class `intstack` using Objective Caml's lists, implementing methods `push`, `pop`, `top` and `size`.
2. Create an instance containing 3 and 4 as stack elements.
3. Define a new class `stack` containing elements answering the method `print : unit -> unit`.
4. Define a parameterized class `['a] stack`, using the same methods.
5. Compare the different classes of stacks.

Delayed Binding

This exercise illustrates how delayed binding can be used in a setting other than subtyping.

Given the program below:

1. Draw the relations between classes.
2. Draw the different messages.
3. Assuming you are in character mode without echo, what does the program display?

```
exception CrLf;;
class chain_read (m) =
  object (self)
  val msg = m
  val mutable res = ""

  method char_read =
    let c = input_char stdin in
      if (c != '\n') then begin
        output_char stdout c; flush stdout
      end;
      String.make 1 c

  method  private chain_read_aux =
      while true do
        let s = self#char_read in
          if s = "\n" then raise CrLf
          else res <- res ^ s;
      done

  method private chain_read_aux2 =
    let s = self#lire_char in
          if s = "\n" then raise CrLf
          else begin res <- res ^ s; self#chain_read_aux2 end

  method chain_read =
    try
      self#chain_read_aux
    with End_of_file  -> ()
      | CrLf -> ()

  method input = res <- ""; print_string msg; flush stdout;
                 self#chain_read

  method get = res
end;;

class mdp_read (m) =
  object (self)
  inherit chain_read m
  method char_read = let c  = input_char stdin in
                     if (c != '\n') then begin
                       output_char stdout '*'; flush stdout
                     end;
```

```
                    let s = " " in s.[0] <- c; s
end;;

let login = new chain_read("Login : ");;
let passwd = new mdp_read("Passwd : ");;
login#input;;
passwd#input;;
print_string (login#get);;print_newline();;
print_string (passwd#get);;print_newline();;
```

Abstract Classes and an Expression Evaluator

This exercise illustrates code factorization with abstract classes.

All constructed arithmetic expressions are instances of a subclass of the abstract class *expr_ar*.

1. Define an abstract class *expr_ar* for arithmetic expressions with two abstract methods: `eval` of type *float*, and `print` of type *unit*, which respectively evaluates and displays an arithmetic expression.
2. Define a concrete class *constant*, a subclass of *expr_ar*.
3. Define an abstract subclass *bin_op* of *expr_ar* implementing methods `eval` and `print` using two new abstract methods `oper`, of type *(float * float) -> float* (used by `eval`) and `symbol` of type *string* (used by `print`).
4. Define concrete classes *add* and *mul* as subclasses of *bin_op* that implement the methods `oper` and `symbol`.
5. Draw the inheritance tree.
6. Write a function that takes a sequence of `Genlex.token`, and constructs an object of type *expr_ar*.
7. Test this program by reading the standard input using the generic lexical analyzer `Genlex`. You can enter the expressions in post-fix form.

The Game of Life and Objects.

Define the following two classes:

- `cell` : for the cells of the world, with the method `isAlive : unit -> bool`
- `world` : with an array of `cell`, and the messages:
  ```
  display : unit -> unit
  nextGen : unit -> unit
  setCell : int * int -> cell -> unit
  getCell : int * int -> cell
  ```

1. Write the class `cell`.

2. Write an abstract class `absWorld` that implements the abstract methods `display`, `getCell` and `setCell`. Leave the method `nextGen` abstract.
3. Write the class `world`, a subclass of `absWorld`, that implements the method `nextGen` according to the growth rules.
4. Write the main program which creates an empty world, adds some cells, and then enters an interactive loop that iterates displaying the world, waiting for an interaction and computing the next generation.

Summary

This chapter described the object extension of the language Objective Caml. The class organization is an alternative to modules that, thanks to inheritance and delayed binding, allows object modeling of an application, as well as reusability and adaptability of programs. This extension is integrated with the type system of Objective Caml and adds the notion of subtype, which allows instances to be used as a subtype in any place where a value of this type is expected. By combining subtyping and delayed binding, we obtain inclusion polymorphism, which, for instance, allows us to build homogeneous lists from the point of view of types, albeit non-homogeneous with regard to behavior.

To Learn More

There are a huge number of publications on object-oriented programming. Each language implements a different model.

A general introduction (still topical for the first part) is "Langages à Objets " ([MNC+91]) which explains the object-oriented approach. A more specialized book, "Langages et modèles à objets" [DEMN98], gives the examples in this domain.

For modeling, the book "Design patterns" ([GHJV95]) gives a catalogue of design patterns that show how reusability is possible.

The reference site for the UML notation is Rational:

Link: http://www.rational.com/uml/resources

For functional languages with an object extension, we mention the "Lisp" objects, coming from the SMALLTALK world, and CLOS (meaning *Common Lisp Object System*), as well as a number of Scheme's implementing generic functions similar to those in CLOS.

Other proposals for object-oriented languages have been made for statically typed functional languages, such as Haskell, a pure functional language which has parameterized and *ad hoc* polymorphism for overloading.

The paper [RV98] presents the theoretical aspects of the object extension of Objective Caml.

To learn more on the static object typing in Objective Caml, you can look at several lectures available online.

Lectures by María-Virginia Aponte:

Link: http://tulipe.cnam.fr/personne/aponte/ocaml.html

A short presentation of objects by Didier Rémy:

Link: http://pauillac.inria.fr/~remy/objectdemo.html

Lectures by Didier Rémy at Magistère MMFAI:

Link: http://pauillac.inria.fr/~remy/classes/magistere/

Lectures by Roberto Di Cosmo at Magistère MMFAI:

Link: http://www.dmi.ens.fr/users/dicosmo/CourseNotes/OO/