B

Objective Caml 3.04

Independently of the development of Objective Caml, several extensions of the language appeared. One of these, named Olabl, was integrated with Objective Caml, starting with version 3.00.

This appendix describes briefly the new features offered in the current version of Objective Caml at the time of this writing, that is. Objective Caml 3.04. This version can be found on the CD-ROM accompanying this book. The new features include:

- *labels*;
- *optional arguments*;
- *polymorphic constructors*;
- the `ocamlbrowser` IDE;
- the `LablTk` library.

The reader is referred to the Objective Caml reference manual for a more detailed description of these features.

Language Extensions

Objective Caml 3.04 brings three language extensions to Objective Caml: labels, optional arguments, and polymorphic constructors. These extensions preserve backward compatibility with the original language: a program written for version 2.04 keeps the same semantics in version 3.04.

Labels

A label is an annotation for the arguments of a function in its declaration and its application. It is presented as a separate identifier of the function parameter (formal or actual), enclosed between an initial symbol `'~'` and a final symbol `':'`.

Labels can appear in the declarations of functions:

Syntax : **let** *f* ~*label*:*p* **=** *exp*

in the anonymous declarations with the keyword **fun** :

Syntax : **fun** ~*label*:*p* **->** *exp*

and in the actual parameter of a function:

Syntax : **(** *f* ~*label*:*exp* **)**

Labels in types The labels given to arguments of a functional expression appear in its type and annotate the types of the arguments to which they refer. (The `'~'` symbol in front of the label is omitted in types.)

```
# let add ~op1:x ~op2:y = x + y ;;
val add : op1:int -> op2:int -> int = <fun>

# let mk_triplet ~arg1:x ~arg2:y ~arg3:z = (x,y,z) ;;
val mk_triplet : arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
```

If one wishes to give the same identifier to the label and the variable, as in `~x:x`, it is unnecessary to repeat the identifier; the shorter syntax `~x` can be used instead.

Syntax : **fun** ~*p* –> *exp*

```
# let mk_triplet ~arg1 ~arg2 ~arg3 = (arg1,arg2,arg3) ;;
val mk_triplet : arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
```

It is not possible to define labels in a declaration of a function by pattern matching; consequently the keyword **function** cannot be used for a function with a label.

```
# let f = function ~arg:x → x ;;
Toplevel input:
#
  let f = function ~arg:x -> x ;;
                   ^^^^^
Syntax error
# let f = fun ~arg:x → x ;;
val f : arg:'a -> 'a = <fun>
```

Labels in function applications When a function is defined with labeled parameters, applications of this function require that matching labels are provided on the function arguments.

```
# mk_triplet ~arg1:'1' ~arg2:2 ~arg3:3.0 ;;
- : char * int * float = '1', 2, 3
# mk_triplet '1' 2 3.0 ;;
- : char * int * float = '1', 2, 3
```

A consequence of this requirement is that the order of arguments having a label does not matter, since one can identify them by their label. Thus, labeled arguments to a function can be "commuted", that is, passed in an order different from the function definition.

```
# mk_triplet ~arg2:2  ~arg1:'1'  ~arg3:3.0 ;;
- : char * int * float = '1', 2, 3
```

This feature is particularly useful for making a partial application on an argument that is not the first in the declaration.

```
# let triplet_0_0 = mk_triplet ~arg2:0 ~arg3:0 ;;
val triplet_0_0 : arg1:'a -> 'a * int * int = <fun>
# triplet_0_0 ~arg1:2 ;;
- : int * int * int = 2, 0, 0
```

Arguments that have no label, or that have the same label as another argument, do not commute. In such a case, the application uses the first argument that has the given label.

```
# let test ~arg1:_ ~arg2:_  _  ~arg2:_  _ = () ;;
val test : arg1:'a -> arg2:'b -> 'c -> arg2:'d -> 'e -> unit = <fun>

# test ~arg2:() ;;  (* the first arg2: in the declaration *)
- : arg1:'a -> 'b -> arg2:'c -> 'd -> unit = <fun>

# test () ;;       (* the first unlabeled argument in the declaration *)
- : arg1:'a -> arg2:'b -> arg2:'c -> 'd -> unit = <fun>
```

Legibility of code Besides allowing re-ordering of function arguments, labels are also very useful to make the function interface more explicit. Consider for instance the `String.sub` standard library function.

```
# String.sub ;;
- : string -> int -> int -> string = <fun>
```

In the type of this function, nothing indicates that the first integer argument is a character position, while the second is the length of the string to be extracted. Objective Caml 3.04 provides a "labelized" version of this function, where the purpose of the different function arguments have been made explicit using labels.

```
# StringLabels.sub ;;
- : string -> pos:int -> len:int -> string = <fun>
```
Clearly, the function `StringLabels.sub` takes as arguments a string, the position of the first character, and the length of the string to be extracted.

Objective Caml 3.04 provides "labelized" versions of many standard library functions in the modules `ArrayLabels`, `ListLabels`, `StringLabels`, `UnixLabels`, and `MoreLabels`. Table B.1 gives the labeling conventions that were used.

label	significance
pos:	a position in a string or array
len:	a length
buf:	a string used as buffer
src:	the source of an operation
dst:	the destination of an operation
init:	the initial value for an iterator
cmp:	a comparison function
mode:	an operation mode or a flag list

Figure B.1: Conventions for labels

Optional arguments

Objective Caml 3.04 allows the definition of functions with labeled *optional* arguments. Such arguments are defined with a default value (the value given to the parameter if the application does not give any other explicitly).

Syntax : **fun ?***name*: **(** *p* **=** *exp*$_1$**)** $\rightarrow$ *exp*$_2$

As in the case of regular labels, the argument label can be omitted if it is identical to the argument identifier:

Syntax : **fun ?(** *name* **=** *exp*$_1$**)** $\rightarrow$ *exp*$_2$

Optional arguments appear in the function type prefixed with the **?** symbol.

```
# let sp_incr ?inc:(x=1) y =  y := !y + x ;;
val sp_incr : ?inc:int -> int ref -> unit = <fun>
```
The function `sp_incr` behaves like the function `incr` from the Pervasives module.
```
# let v = ref 4  in  sp_incr v ; v ;;
- : int ref = {contents = 5}
```
However, one can specify a different increment from the default.
```
# let v = ref 4  in  sp_incr ~inc:3 v ; v ;;
- : int ref = {contents = 7}
```

A function is applied by giving the default value to all the optional parameters until the actual parameter is passed by the application. If the argument of the call is given without a label, it is considered as being the first non-optional argument of the function.

```
# let f ?(x1=0) ?(x2=0) x3 x4 = 1000*x1+100*x2+10*x3+x4 ;;
val f : ?x1:int -> ?x2:int -> int -> int -> int = <fun>
# f 3 ;;
- : int -> int = <fun>
# f 3 4 ;;
- : int = 34
# f ~x1:1 3 4 ;;
- : int = 1034
# f ~x2:2 3 4 ;;
- : int = 234
```

An optional argument can be given without a default value, in this case it is considered in the body of the function as being of the type *'a option*; `None` is its default value.

Syntax : **fun ?***name*:*p* -> *exp*

```
# let print_integer ?file:opt_f n =
    match opt_f with
        None → print_int n
      | Some f → let fic = open_out f in
                 output_string fic (string_of_int n) ;
                 output_string fic "\n" ;
                 close_out fic ;;
val print_integer : ?file:string -> int -> unit = <fun>
```

By default, the function `print_integer` displays its argument on standard output. If it receives a file name with the label `file`, it outputs its integer argument to that file instead.

Note

If the last parameter of a function is optional, it will have to be applied explicitly.

```
# let test ?x ?y  n  ?a ?b  =  n ;;
val test : ?x:'a -> ?y:'b -> 'c -> ?a:'d -> ?b:'e -> 'c = <fun>
# test 1 ;;
- : ?a:'_a -> ?b:'_b -> int = <fun>
# test 1 ~b:'x' ;;
- : ?a:'_a -> int = <fun>
# test 1 ~a:() ~b:'x' ;;
- : int = 1
```

Labels and objects

Labels can be used for the parameters of a method or an object's constructor.

```
# class point ?(x=0) ?(y=0) (col : Graphics.color) =
    object
      val pos = (x,y)
      val color = col
      method print ?dest:(file=stdout) () =
          output_string file "point (" ;
          output_string file (string_of_int (fst pos)) ;
          output_string file "," ;
          output_string file (string_of_int (snd pos)) ;
          output_string file ")\n"
    end ;;
class point :
  ?x:int ->
  ?y:int ->
  Graphics.color ->
  object
    method print : ?dest:out_channel -> unit -> unit
    val color : Graphics.color
    val pos : int * int
  end

# let obj1 = new point ~x:1 ~y:2 Graphics.white
  in obj1#print () ;;
point (1,2)
- : unit = ()
# let obj2 = new point Graphics.black
  in obj2#print () ;;
point (0,0)
- : unit = ()
```

Labels and optional arguments provide an alternative to method and constructor overloading often found in object-oriented languages, but missing from Objective Caml.

This emulation of overloading has some limitations. In particular, it is necessary that at least one of the arguments is not optional. A dummy argument of type *unit* can always be used.

```
# class number ?integer ?real () =
    object
      val mutable value = 0.0
      method print = print_float value
      initializer
        match (integer,real) with
            (None,None) | (Some _,Some _) → failwith "incorrect number"
          | (None,Some f) → value <- f
```

```
          | (Some n,None) → value <- float_of_int n
    end ;;
class number :
  ?integer:int ->
  ?real:float ->
  unit -> object method print : unit val mutable value : float end

# let n1 = new number ~integer:1 () ;;
val n1 : number = <obj>
# let n2 = new number ~real:1.0 () ;;
val n2 : number = <obj>
```

Polymorphic variants

The variant types of Objective Caml have two principal limitations. First, it is not possible to extend a variant type with a new constructor. Also, a constructor can belong to only one type. Objective Caml 3.04 features an alternate kind of variant types, called *polymorphic variants* that do not have these two constraints.

Constructors for polymorphic variants are prefixed with a **‘** (backquote) character, to distinguish them from regular constructors. Apart from this, the syntactic constraints on polymorphic constructors are the same as for other constructors. In particular, the identifier used to build the constructor must begin with a capital letter.

Syntax : **‘***Name*

ou

Syntax : **‘***Name type*

A group of polymorphic variant constructors forms a type, but this type does not need to be declared before using the constructors.

```
# let x = ‘Integer 3 ;;
val x : [> ‘Integer of int] = ‘Integer 3
```

The type of `x` with the symbol **[>** indicates that the type contains at least the constructor ‘`Integer` *`int`*.

```
# let int_of = function
      ‘Integer n → n
    | ‘Real r → int_of_float r ;;
val int_of : [< ‘Integer of int | ‘Real of float] -> int = <fun>
```

Conversely, the symbol **[<** indicates that the argument of `int_of` belongs to the type that contains at most the constructors ‘`Integer` *`int`* and ‘`Real` *`float`*.

It is also possible to define a polymorphic variant type by enumerating its constructors:

Syntax : **type** *t* **= [** ‘*Name*$_1$ **|** ‘*Name*$_2$ **|** ... **|** ‘*Name*$_n$ **]**

or for parameterized types:

Syntax : **type** *('a,'b,...) t* **= [** ‘*Name*$_1$ **|** ‘*Name*$_2$ **|** ... **|** ‘*Name*$_n$ **]**

```
# type value = [ ‘Integer of int | ‘Real of float ] ;;
type value = [ ‘Integer of int | ‘Real of float]
```

Constructors of polymorphic variants can take arguments of different types.

```
# let v1 = ‘Number 2
  and v2 = ‘Number 2.0 ;;
val v1 : [> ‘Number of int] = ‘Number 2
val v2 : [> ‘Number of float] = ‘Number 2
```

However, v1 and v2 have different types.

```
# v1=v2 ;;
Toplevel input:
#
  v1=v2 ;;
     ^^
This expression has type [> ‘Number of float] but is here used with type
  [> ‘Number of int]
```

More generally, the constraints on the type of arguments for polymorphic variant constructors are accumulated in their type by the annotation **&**.

```
# let test_nul_integer = function ‘Number n → n=0
  and test_nul_real = function ‘Number r → r=0.0 ;;
val test_nul_integer : [< ‘Number of int] -> bool = <fun>
val test_nul_real : [< ‘Number of float] -> bool = <fun>
# let test_nul x = (test_nul_integer x) || (test_nul_real x) ;;
val test_nul : [< ‘Number of float & int] -> bool = <fun>
```

The type of test_nul indicates that the only values accepted by this function are those with the constructor ‘Number and an argument which is at the same time of type *int* and of *float*. That is, the only acceptable values are of type *'a*!

```
# let f () = test_nul (failwith "returns a value of type 'a") ;;
val f : unit -> bool = <fun>
```

The types of the polymorphic variant constructor are themselves likely to be polymorphic.

```
# let id = function ‘Ctor → ‘Ctor ;;
val id : [< ‘Ctor] -> [> ‘Ctor] = <fun>
```

The type of the value returned from id is “the group of constructors that contains at least ‘Ctor” therefore it is a polymorphic type which can instantiate to a more precise type. In the same way, the argument of id is “the group of constructors that contains

no more than ‘Ctor” which is also likely to be specified. Consequently, they follow the general polymorphic type mechanism of Objective Caml knowing that they are likely to be weakened.

```
# let v = id ‘Ctor ;;
val v : _[> ‘Ctor] = ‘Ctor
```

v, the result of the application is not polymorphic (as denoted by the character _ in the name of the type variable).

```
# id v ;;
- : _[> ‘Ctor] = ‘Ctor
```

v is monomorphic and its type is a sub-type of “contains at least the constructor ‘Ctor”. Applying it with id will force its type to be a sub-type of“contains no more than the constructor ‘Ctor”. Logically, it must now have the type “contains exactly ‘Ctor”. Let us check.

```
# v ;;
- : [ ‘Ctor] = ‘Ctor
```

As with object types, the types of polymorphic variant constructors can be open.

```
# let is_integer = function
      ‘Integer (n : int) → true
    | _ → false ;;
val is_integer : [> ‘Integer of int] -> bool = <fun>
# is_integer (‘Integer 3) ;;
- : bool = true
# is_integer ‘Other ;;
- : bool = false
```

All the constructors are accepted, but the constructor ‘Integer must have an integer argument.

```
# is_integer (‘Integer 3.0) ;;
Toplevel input:
#
  is_integer (‘Integer 3.0) ;;
              ^^^^^^^^^^^^
This expression has type [> ‘Integer of float] but is here used with type
  [> ‘Integer of int]
```

As with object types, the type of a constructor can be cyclic.

```
# let rec long = function ‘Rec x → 1 + (long x) ;;
val long : ([< ‘Rec of ’a] as ’a) -> int = <fun>
```

Finally, let us note that the type can be at the same time a sub-group and one of a group of constructors. Starting with a a simple example:

```
# let ex1 = function ‘C1 → ‘C2 ;;
val ex1 : [< ‘C1] -> [> ‘C2] = <fun>
```

Now we identify the input and output types of the example by a second pattern.

```
# let ex2 = function ‘C1 → ‘C2 | x → x ;;
val ex2 : ([> ‘C2 | ‘C1] as ’a) -> ’a = <fun>
```

We thus obtain the open type which contains at least ‘C2 since the return type contains at least ‘C2.

```
# ex2 ( ‘C1 : [> ‘C1 ] ) ;;  (* is a subtype of [<‘C2|‘C1| .. >‘C2] *)
```

```
- : _[> `C2 | `C1] = `C2
# ex2 ( `C1 : [ `C1 ] ) ;;   (* is not a subtype of [<`C2|`C1| .. >`C2] *)
Toplevel input:
# ex2 ( `C1 : [ `C1 ] ) ;;   (* is not a subtype of [<`C2|`C1| .. >`C2] *)
        ^^^
This expression has type [ `C1] but is here used with type [> `C2 | `C1]
```

`LablTk` *Library*

The interface to Tcl/Tk was integrated in the distribution of Objective Caml 3.04, and is available for Unix and Windows. The installation provides one new command: `labltk`, which launches a toplevel interactive loop integrating the `LablTk` library.

The `LablTk` library defines a large number of modules, and heavily uses the language extensions of Objective Caml 3.04. A detailed presentation of this module falls outside the scope of this appendix, and we invite the interested reader to refer to the documentation of Objective Caml 3.04.

There is also an interface with Gtk, written in class-based style, but it is not yet part of the Objective Caml distribution. It should be compatible with Unix and Windows.

OCamlBrowser

OcamlBrowser is a code browser for Objective Caml, providing a `LablTk`-based graphical user interface. It integrates a "navigator" allowing to browse various modules, to look at their contents (names of values and types), and to edit them.

When launching OCamlBrowser by the command `ocamlbrowser`, the list of all the compiled modules available (see figure B.2) is displayed. One can add more modules by specifying a path to find them. From the menu `File`, one can launch a toplevel interactive loop or an editor in a new window.

When one of the modules is clicked on, a new window opens to display its contents (see figure B.3). By selecting a value, its type appears in bottom of the window.

In the main window, one can search on the name of a function. The result appears in a new window. The figure B.4 shows the result of a search on the word `create`.

There are other possibilities that we let the user discover.

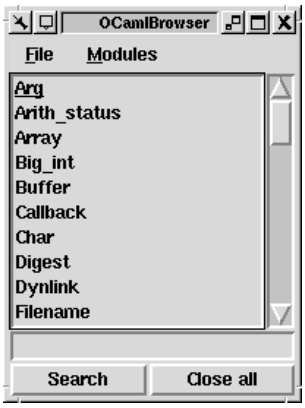

Figure B.2: OCamlBrowser : the main window

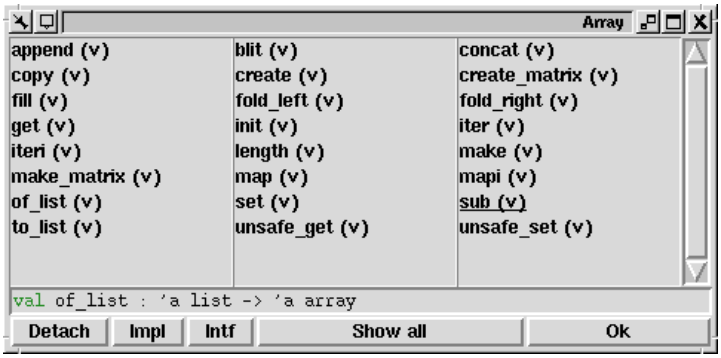

Figure B.3: OCamlBrowser : module contents

Figure B.4: OCamlBrowser : search for `create`