Introduction to the OCaml programming language

Frédéric Blanqui (INRIA)

13 March 2013

These notes have been written for a 7-days school organized at the Institute of Applied Mechanics and Informatics (IAMA) of the Vietnamese Academy of Sciences and Technology (VAST) at Ho Chi Minh City, Vietnam, from Tuesday 12 March 2013 to Tuesday 19 March 2013. The mornings were dedicated to theoretical lectures introducing basic notions in mathematics and logic for the analysis of computer programs [1]. The afternoons were practical sessions introducing the OCaml programming language (these notes) and the Coq proof assistant (notes in [2]).

OCaml is a typed programming language with features that make it very well adapted for symbolic computation:

- It is a functional programming language: functions are first-class objects; they can be returned and passed as arguments.
- It is strongly typed: pure functional programs cannot have runtime errors (except when computations exceed the memory).
- One can define inductive data types and define functions by using pattern matching.
- Functions can be polymorphic (the same code can be used for different types).
- OCaml automatically manages the memory (using a garbage collector) so that developers do not have to manipulate pointers, an important source of bugs.
- It has an efficient compiler.
- It also provides other features like modules, objects, ...

The development of OCaml started in 1996. It is developed at INRIA, France. You can find some short history of OCaml on its web page.

1 Before starting

I suggest to use a computer running the Linux operating system, *e.g.* the Ubuntu distribution (but OCaml can also be used under Windows). Then, you have to install the following software:

- emacs: a text editor
- ocaml: the OCaml compiler
- tuareg-mode: an Emacs mode for editing OCaml programs
- ledit: a line editor for text-based interactive programs

In Ubuntu, the installation is easy: run the "Ubuntu software center", search for these programs and click on "Install".

Finally, add the following line in your `~/.bashrc` file:

```
alias ocaml='ledit -x -h ~/.ocaml-history ocaml'
```

2 Emacs

Here is a number of useful basic shortcuts for Emacs:

- To interrupt a shorcut: Ctrl-g
- To open a file in a buffer: Ctrl-x Ctrl-f
- To save a file: Ctrl-x s
- To close a buffer: Ctrl-x k
- To undo a change: Ctrl-underscore
- To copy some text: Ctrl-space, move cursor, Alt-w
- To paste some text: Ctrl-y
- To remove some text: Ctrl-space, move cursor, Ctrl-w
- To search a string: Ctrl-s
- To replace a string by another one: Esc-%
- To indent: Esc-q
- To expand a string already written: Esc-/
- To increase font size: Ctrl-x Ctrl-+
- To decrease font size: Ctrl-x Ctrl–

3 OCaml types and values

The OCaml language has a compiler (in fact two) but it also has an interpreter that can be run interactively (command `ocaml`) for trying some simple code. We will use this at the beginning to discover the basic features of OCaml. Do Ctrl-c to interrupt `ocaml` and Ctrl-d to exit.

Some builtin data types and their values:

- `int`: integers between `min_int` and `max_int` (-2^{30} and $2^{30}-1$ on 32-bits architectures)
- `bool`: `true`, `false`
- `char`: `'a'`, `'b'`, `'\n'` (new line), ...
- `string`: `"hello!"`, ...
- `float`: `1.2`, ...
- `unit`: singleton type with only one value, `()`, used for functions changing the memory, or reading or printing something, but returning no result

The type for functions taking values of type `t` as argument and returning a value of type `u` is `t -> u`. The type `t1 -> t2 -> u` denotes `t1 -> (t2 -> u)` (type of functions taking two arguments of type `t1` and `t2` respectively and returning a value of type `u`) and not `(t1 -> t2) -> u` (type of functions taking an argument of type `t1 -> t2` and returning a value of type `u`). We say that `->` associates to the right.

The type for tuples `(x1,...,xn)` where `x1` is of type `t1`, ..., `xn` is of type `tn`, is `t1 *...* tn`. For instance, the type for pairs of integers is `int * int`.

Some builtin functions:

- polymorphic comparison functions: `=`, `<`, `<=`, `>`, `>=` : `'a -> 'a -> bool` (the type variable `'a` denotes any type)
- arithmetic operations: `+`, `-`, `*` : `int -> int -> int`
- boolean operators: `&&` (and), `||` (or) : `bool -> bool -> bool`, `not` : `bool -> bool`
- print an `int` on `stdout` (standard output): `print_int` : `int -> unit`
- print a `string` on `stdout`: `print_string` : `string -> unit`
- conditional: `if` ... `then` ... `else` ... : `bool -> 'a -> 'a -> 'a`
- sequence: `;` : `unit -> 'a -> 'a`. *instruction*$_1$ `;` *instruction*$_2$ executes *instruction*$_1$, then executes *instruction*$_2$ and returns its result. i_1 `;` i_2 `;` i_3 is the same as i_1 `;` `(` i_2 `;` i_3 `)` (`;` associates to the right).

How to define constants and non-recursive functions?

`let` $function\_name\ argument_1 \ldots argument_n$ `=` $definition$ `;;`

For a recursive function:

`let rec` $function\_name\ argument_1 \ldots argument_n$ `=` $definition$ `;;`

Note that all names occurring in $definition$ must be already defined, except $function\_name$ when it is a recursive definition. Note then that the order of definitions is important.

Texts starting with `(*` and ending with `*)` are comments ignored by the compiler. They can be added anywhere and can be nested.

Exercise 1 Try some expressions in `ocaml` and see what are the results. For instance:

```
1;;
1+3;;
true+3;; (* Typing error *)

2*1+3;;
2*(1+3);;
(2*1)+3;;

letx : bool = true;; (* Syntax error: [letx] with no space between
 [let] and [x] is understood as a name *)

let x : bool = true;;
let x : int = 1;; (* We can redefine [x]. *)

let add_5 (x : int) = x + 5;;
add_5 2;;

(* Types are not necessary here since they can be inferred by
 OCaml. So we can in fact write: *)

let x = true;;
let x = 1;;
let add_5 x = x + 5;;

let y = a + 1;; (* [a] is undefined. *)

let y = 1;;
let y = y+1;; (* The [y] occurring in the definition of [y] refers to
 the previously defined [y].*)
```

```
let rec fact n = if n <= 1 then 1 else n * fact (n-1);;

1; 2;;
ignore 1; 2;;

(1,true);;
```

4 Compiling OCaml programs

Writing and compiling an OCaml program:

1. Declare the types (interface) of exported functions in a file `my_prog.mli`:

   ```
   val fact : int -> int;;
   ```

2. Compile the interface with:

   ```
   ocamlc -c my_prog.mli
   ```

   This creates the file `my_prog.cmi`.

3. Write the implementation of the declared functions in the file `my_prog.ml`:

   ```
   let rec fact n = if n <= 1 then 1 else n * fact (n-1);;
   ```

4. Compile the implementation with:

   ```
   ocamlc -c my_prog.ml
   ```

   This creates the file `my_prog.cmo`.

5. Build an executable program with:

   ```
   ocamlc -o my_prog my_prog.cmo
   ```

   This creates the executable file `my_prog`.

6. Execute your program with:

   ```
   ./my_prog
   ```

Separating interfaces and implementations are many advantages. For instance:

- This allows separate development and compilation. One can write a program `my_prog2.ml` using the function `fact` without having the file `my_prog.ml` but only its interface `my_prog.mli`. Even if `my_prog.ml` is modified, `my_prog2.ml` does not need to be recompiled as long as the interface `my_prog.mli` is unchanged.
- This allows to hide some implementation details. A type does not need to be defined in the interface (we then say that this is an abstract data type). To define some function, one can use auxiliary functions that do not need to be exported.

All this can be automated by using a `Makefile` and the make tool. A `Makefile` describes the dependencies between files and the commands to execute (preceded by a tabulation) to build a file from its dependencies. Here is a possible `Makefile` for `my_prog` (a more generic `Makefile` could be written using the `ocamldep` tool described in the OCaml documentation):

```
default: my_prog

my_prog: my_prog.cmo
        ocamlc -o my_prog my_prog.cmo

my_prog.cmo: my_prog.ml my_prog.cmi
        ocamlc -c my_prog.ml

my_prog.cmi: my_prog.mli
        ocamlc -c my_prog.mli
```

Note that if there is no interface, then all functions are exported and every thing can be done with:

```
ocamlc -o my_prog my_prog.ml
```

Compilation in Emacs:

1. do Ctrl-x s to save your file `my_prog.ml`
2. do Ctrl-c Ctrl-c and write `ocamlc -o my_prog my_prog.ml` as compilation command
3. in case of a compilation error, do Ctrl-x backquote to jump to the next error
4. in a terminal, run your program by typing `./my_prog`

5 Inductive types and pattern-matching

```
(* A very nice feature of OCaml is its ability to define inductive
   types and use pattern-matching for defining functions. As an
```

```
   example, consider the inductive type [seq] of sequences of
   integers. A value of type [seq] is either [Empty] for the empty
   sequence, or [Add (x,l)] for adding an int [x] at the beginning of
   an already built sequence [l]. *)

type seq = Empty | Add of int * seq;;

(* Examples of sequences. *)

let s123 = Add (1, Add (2, Add (3, Empty)));;
let s456 = Add (4, Add (5, Add (6, Empty)));;

(* Function printing a sequence. Elements are separated by the string
   [sep]. We use pattern-matching for defining functions on
   sequences. Note that the order of patterns is important. *)

let rec print_seq sep l =
  match l with
    | Empty -> ()
    | Add (x, Empty) -> print_int x
    | Add (x, l') -> (* The order of patterns is important! *)
      print_int x; print_string sep; print_seq sep l';;

let print_seq_nl sep l = print_seq sep l; print_newline();;

(* Function computing the concatenation of two sequences. *)

let rec concat l1 l2 =
  match l1 with
    | Empty -> l2
    | Add (x, l1') -> Add (x, concat l1' l2);;

print_seq_nl ", " (concat s123 s456);;

(* Function adding an element at the end of a list. *)

let rec add_at_the_end x l =
  match l with
    | Empty -> Add (x, Empty)
    | Add (y, l') -> Add (y, add_at_the_end x l');;

(* Function putting the elements of a list in reverse order. *)

let rec reverse l =
  match l with
    | Empty -> Empty
```

```
    | Add (x, l') -> add_at_the_end x (reverse l');;

print_seq_nl ", " (reverse s123);;

(* Function inserting an element in a sorted list. *)

let rec insert x l =
  match l with
    | Empty -> Add (x, Empty)
    | Add (y, l') ->
      if x > y then Add (y, insert x l') else Add (x, l);;

(* Function sorting the elements of a liste in increasing order. *)

let rec sort l =
  match l with
    | Empty -> Empty
    | Add (x, l') -> insert x (sort l');;

print_seq_nl ", " (sort (concat s456 s123));;
```

6 Functions as arguments and results

```
(* In OCaml, functions can take functions as argument and return
   functions as result. Example: a more generic sorting function taking
   as argument a comparison function [cmp : int -> int -> int] such that:

   [cmp x y < 0] if [x] is smaller than [y]
            =              equal to
            >              greater than *)

let rec insert cmp x l =
  match l with
    | Empty -> Add (x, Empty)
    | Add (y, l') ->
      if cmp x y > 0 then Add (y, insert cmp x l') else Add (x, l);;

let rec sort cmp l =
  match l with
    | Empty -> Empty
    | Add (x, l') -> insert cmp x (sort cmp l');;

let sort_incr = sort (fun x y -> if x < y then -1 else if x = y then 0 else 1);;

let sort_decr = sort (fun x y -> if x < y then 1 else if x = y then 0 else -1);;
```

```
print_seq_nl ", " (sort_incr (concat s456 s123));;
print_seq_nl ", " (sort_decr (concat s456 s123));;
```

7 Polymorphism

```
(* In fact, all the functions define previously on [int] would work as
   well with any other type. This can be done in OCaml by using a
   polymorphic definition of [seq] as follows, where ['a] denotes a
   type variable. *)

type 'a seq = Empty | Add of 'a * 'a seq;;

(* Examples of lists of integers, strings, pairs and functions. *)

let s123 = Add (1, Add (2, Add (3, Empty)));;
let s456 = Add (4, Add (5, Add (6, Empty)));;

let sabc = Add ("a", Add ("b", Add ("c", Empty)));;

let seq_pairs = Add (("a",1), Add (("b",2), Add (("c",3), Empty)));;

let seq_funs = Add ((fun x -> x+1), Add ((fun x -> x+2), Empty));;

(* Then, we only have to generalize [print_seq] by adding a parameter
   [print_elt] for printing elements. *)

let rec print_seq print_elt sep l =
  match l with
    | Empty -> ()
    | Add (x, Empty) -> print_elt x
    (* beware: the order of patterns is important! *)
    | Add (x, l') ->
      print_elt x; print_string sep; print_seq print_elt sep l';;

let print_seq_nl print_elt sep l =
  print_seq print_elt sep l; print_newline();;

print_seq_nl print_int ", " (concat s123 s456);;
```

8 Going further

You will find a lot of material on OCaml on its web page http://ocaml.inria.fr including:

- The reference manual.
- Tutorials.
- The standard library.
- Links towards many libraries and software developed in OCaml.

Books on OCaml and programming in OCaml: [3, 4, 5]. (More books and lecture notes are available in French.)

There are also a number of useful mailing lists:

- caml-announce@inria.fr
- ocaml_beginners@yahoogroups.com
- caml-news-weekly@lists.idyll.org
- caml-list@inria.fr
- ocaml-jobs@inria.fr

References

[1] F. Blanqui. Elements of mathematics and logic for the analysis of computer programs, 2013. Lecture notes, 37 pages.

[2] F. Blanqui. Introduction to the Coq proof assistant, 2013. Lecture notes, 6 pages.

[3] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[4] G. Cousineau and M. Mauny. *The functional approach to programming*. Cambridge University Press, 1998.

[5] Jon D. Harrop. *OCaml for scientists*. Flying Frog Consultancy Ltd, 2005.