Introduction to Objective Caml

Stefan Wehr

University of Freiburg.

October 30, 2006

Why OCaml?

- Convenient encoding of tree structures
- Powerful pattern matching facilities
- Close correspondence to mathematical notation

Features of OCaml

- Functional language (functions are first-class values)
- Strong and statically typed
- Parametric polymorphism
- Type inference
- Recursive, algebraic datatypes (trees, lists, . . .)
- Garbage collection
- Modul-system
- (Object-system)

Applications written in OCaml

- File sharing: MLdonkey (`http://mldonkey.org/`)
- File synchronizer: unison (`http://www.cis.upenn.edu/~bcpierce/unison/`)
- Compilers and interpreters: OCaml, XQuery, XDuce, CDuce
- Proof assistant: Coq (`http://coq.inria.fr/`)

The Toplevel Loop

- Interactive development
- Evaluation of expressions (calculator)
- Definitions

```
$ ocaml
        Objective Caml version 3.09.2

# 39 + 3;;
- : int = 42
# let answer = 39 + 3;;
val answer : int = 42
#
```

Basic Types (1)

```
# ();;
- : unit = ()
```

- Singleton type: `()` is the only element of `unit`
- Similar to `void` in C or Java
- Result type of functions with side effects

```
# 2 + 5 * 8;;
- : int = 42
```

- Signed integers, represented by a machine word minus one bit
- Common Operators: `+` , `-`, `*`, `/`, `mod`
- Conversions: `string_of_int`, `int_of_string`, `float_of_int`

Basic Types (2)

```
# 3.1415926536 *. 2.0;;
- : float = 6.2831853072
```

- IEEE double-precision floating point, equivalent to C's double
- Arithmetic operators end with a dot: `+.` , `-.`, `*.`, `/.`
- Conversions: `string_of_float`, `int_of_float`

```
# Char.uppercase 'x';;
- : char = 'X'
```

- Latin-1 characters (unicode library: `http://camomile.sf.net/`)
- Functions: `Char.lowercase`, `Char.uppercase`
- Conversions: `Char.code` (character → integer),
  `Char.chr` (integer → character)

Basic Types (3)

```
# "Hello " ^ "World\n";;
- : string = "Hello World\n"
```

- Strings with Latin-1 encoding
- Operators: `^` (concatenation), `"Hello".[1]` (index access)
- Functions: `String.length`, `String.sub`

```
#  1 = 2 || false;;
- : bool = false
```

- Operators: `&&`, `||`, `not`
- Comparisons: = (equality), <> (inequality), <, <=, >, >=
  These operators work on arbitrary but equal types; for some types, a runtime exception is raised.

Conditionals and Variables

Conditionals

```
# if 1 < 2 then 3 + 7
  else (if "Hello" = "stefan" then 0 else 42);;
- : int = 10
```

Variables

- Variables are *names* for values
- No assignment!

```
# let x = 4;;
val x : int = 4
# 38 + x;;
- : int = 42
# let y = 3 in 39 + y;;
- : int = 42
# y;;
Unbound value y
```

Functions

```
# let square x = x * x ;;
val square : int -> int = <fun>
# square 42;;
- : int = 1764
```

- Function type: `t1 -> t2`
- Function call without parenthesis around argument

```
# let average x y = (x + y) / 2;;
val average : int -> int -> int = <fun>
# average 21 63;;
- : int = 42
```

- Type of multi-argument functions: `t1 -> t2 -> ... -> tn`
- Function call: concatenate all arguments to the function

Nested Functions

- Functions may be arbitrarily nested.

```
# let sum_of_3 x y z =
    let sum a b = a + b
     in sum x (sum y z);;
val sum_of_3 : int -> int -> int -> int = <fun>
# sum_of_3 1 2 3;;
- : int = 6
```

Recursive Functions

- A recursive function calls itself inside its own body.
- Defined as ordinary functions, but uses `let rec` instead of `let`.
- Example: function that computes x^i

```
# let rec power i x =
    if i = 0 then
       1.0
    else
      x *. (power (i - 1) x);;
val power : int -> float -> float = <fun>
# power 5 2.0;;
- : float = 32.
```

Mutually Recursive Functions

- Connect several `let rec` definitions with the keyword `and`.

```
# let rec f i j =
    if i = 0 then
      j
    else g (j - 1)
  and g j =
    if j mod 3 = 0 then
      j
    else f (j - 1) j;;
val f : int -> int -> int = <fun>
val g : int -> int = <fun>
# g 5;;
- : int = 3
```

Polymorphic Functions

- Work on values of arbitrary type
- Arbitrary types represented as type variables `'a`, `'b`, ...

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

The Value Restriction

- Only values can be polymorphic.
- Function applications are not values.
- The value restriction is needed to ensure soundness in the presence of side-effects.

```
# let id' = id id;;
val id' : '_a -> '_a = <fun>
# id' 5;;
- : int = 5
# id';;
- : int -> int = <fun>
```

Higher-order functions

- Functions are ordinary values.
- A higher-order function takes another function as an argument or returns it as the result.
- Partial application of a function (with less arguments than expected) returns another function

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# let inc = add 1;;
val inc : int -> int = <fun>
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# compose inc inc 0;;
- : int = 2
```

Function Types in Detail

- The arrow associates to the right: The type `int -> int -> int` is the same as `int -> (int -> int)`
- `add` takes an `int` and returns a function of type `int -> int`
- Function application associates to the left: The expression `add 1 2` is the same as `(add 1) 2`
- The sub-expression `(add 1)` has type `int -> int` so we can apply it to the integer `2`

```
# let inc = add 1;;
val inc : int -> int = <fun>
# inc 2;;
- : int = 3
```

Anonymous Functions

- The keyword `fun` constructs an anonymous function.

```
# fun x -> x + 1;;
- : int -> int = <fun>
# compose inc (fun x -> x + 1) 0;;
- : int = 2
```

- Definitions such as `let add x y = x + y` are just syntactic sugar. Here is the expanded definition:

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

Simple Pattern Matching

- Powerful feature
- Defines expressions by case analysis
- Simple pattern: constant or variable

  Constant matches only the constant value given

  Variable matches all values and binds the value to the variable

```
# let rec fib i =
    match i with
          0 -> 0
        | 1 -> 1
        | j -> fib (j - 2) + fib (j - 1);;
val fib : int -> int = <fun>
# fib 1;;
- : int = 1
# fib 6;;
- : int = 8
```

Matching Order

- Cases of a `match` expression are tried in sequence, from top to bottom.
- The body of the first matching case is evaluated.
- The following definition of `fib` is wrong (`fib` loops forever when called).

```
# let rec fib i =
    match i with
        j -> fib (j - 2) + fib (j - 1)
      | 0 -> 0
      | 1 -> 1;;
Warning U: this match case is unused.
Warning U: this match case is unused.
val fib : int -> int = <fun>
# fib 5;;
Stack overflow during evaluation (looping recursion?).
```

Incomplete Matches

OCaml issues a warning if the cases of a `match` do not cover all possible values:

```
# let rec fib i =
    match i with
        0 -> 0
      | 1 -> 1;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
2
val fib : int -> int = <fun>
# fib 2;;
Exception: Match_failure ("", 55, 2).
```

Functions with Matching

- Common situation: pattern matching on the last argument of a function
- Use the `function` keyword instead of an explicit `match` expression

```
# let rec mult x = function
      0 -> 0
    | y -> x + mult x (y - 1);;
val mult : int -> int -> int = <fun>
# mult 1 2;;
- : int = 2
# mult 3 2;;
- : int = 6
```

Matching Characters

```
# let is_uppercase = function
      'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
    | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
    | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U'
    | 'V' | 'W' | 'X' | 'Y' | 'Z' -> true
    | c                           -> false;;
val is_uppercase : char -> bool = <fun>
# is_uppercase 'A';;
- : bool = true
# is_uppercase 'a';;
- : bool = false
```

With pattern ranges and wildcard pattern:

```
# let is_uppercase = function
      'A' .. 'Z' -> true
    | _          -> false;;
val is_uppercase : char -> bool = <fun>
```

Matching Strings

```
# let hall_of_fame = function
      "Adel" -> "Sellimi"
    | "Rodolfo" -> "Cardoso"
    | "Altin" -> "Rraklli"
    | "Harry" -> "Decheiver"
    | "Ali" -> "Günes"
    | "Uwe" -> "Wassmer"
    | _ -> "?"
```

Patterns Everywhere

- Patterns are used in all binding mechanisms:

  `let` *pattern* `=` *expression*

  `let` *name pattern . . . pattern* `=` *expression*

  `fun` *pattern* `->` *expression*

- Very useful with tuples and records (introduced next)

Tuples

- Fixed-length sequences of values with arbitrary types
- Construction:

```
# let p = ("2-times", (fun x -> x * 2), 2 = 42);;
val p : string * (int -> int) * bool
      = ("2-times", <fun>, false)
```

- Elimination by pattern matching:

```
# let (a, b, c) = p;;
val a : string = "2-times"
val b : int -> int = <fun>
val c : bool = true
# match p with (a, _, _) -> a;;
- : string = "2-times"
```

- Pairs can be eliminated with `fst` and `snd`:

```
# fst (1,2);;
- : int = 1
# snd (1,2);;
- : int = 2
```

Lists

- Variable-length sequences of values with the same type
- Two constructors:

  Nil `[]`, the empty list

  Cons $e_1 :: e_2$, creates a new list with first element e_1 and rest of the list e_2
- Shorthand notation:
  $[e_1; \ldots; e_n]$ is identical to $e_1 :: (e_2 :: \ldots :: (e_n :: [\,]) \ldots)$
- `t list` is the type of lists with elements of type `t`

```
# let l = "Hello" :: "World" :: [];;
val l : string list = ["Hello"; "World"]
# let l' = [1;2;3];;
val l' : int list = [1; 2; 3]
```

Lists and Pattern Matching

Lists are eliminated using pattern matching:

```
# let rec inc_list = function
      []      -> []
    | i :: l -> (i + 1) :: inc_list l;;
val inc_list : int list -> int list = <fun>
# inc_list [1; 2; 3; 4];;
- : int list = [2; 3; 4; 5]
# let rec sum_list = function
      []      -> 0
    | i :: l -> i + sum_list l;;
val sum_list : int list -> int = <fun>
# sum_list [1; 2; 3; 4];;
- : int = 10
```

The Map Function

- The function `List.map` applies a function to every element in a list
- `List.map : ('a -> 'b) -> 'a list -> 'b list`
- We can define the function `inc_list` in terms of map because `inc_list [i1; ...; in]` = `[i1+1; ...; in+1]`.

```
# let inc_list = List.map (fun i -> i+1);;
val inc_list : int list -> int list = <fun>
# inc_list [1; 2; 3; 4];;
- : int list = [2; 3; 4; 5]
```

The Fold Function

- The function `List.fold_right` "folds" a function over a list
- `List.fold_right :`
  `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`
- We can define the function `sum_list` in terms of map because `sum_list [i1; ...; in]` = `i1 + ... + in`.

```
# let sum_list l = List.fold_right (+) l 0;;
val sum_list : int list -> int = <fun>
# sum_list [1; 2; 3; 4];;
- : int = 10
```

Algebraic datatypes

- They represent the union of several different types.
- Every alternative has an unique, explicit name.
- General syntax:

  `type` *typename* `=`

  $\quad\quad$ Name_1 `of` type_1

  $\quad$ | Name_2 `of` type_2

  $\quad\quad\vdots$

  $\quad$ | Name_n `of` type_n

- The names Name_i are called constructors; they must start with a capital letter.
- The part `of` type_i is optional.

Example

```
# type number =
      Zero
    | Integer of int
    | Fraction of (int * int);;
type number = Zero | Integer of int
            | Fraction of (int * int)
# Zero;;
- : number = Zero
# Integer 1;;
- : number = Integer 1
# let semi = Fraction (1, 2);;
val semi : number = Fraction (1, 2)
```

Pattern Matching with Algebraic Datatypes

```
# let float_of_number = function
      Zero
        -> 0.0
    | Integer i
        -> float_of_int i
    | Fraction (i,j)
        -> float_of_int i /. float_of_int j;;
val float_of_number : number -> float = <fun>
# float_of_number semi;;
- : float = 0.5
```

Binary Trees

```
# type 'a tree = Node of ('a * 'a tree * 'a tree) | Leaf;;
type 'a tree = Node of ('a * 'a tree * 'a tree) | Leaf
# let rec insert x = function
      Leaf -> Node (x, Leaf, Leaf)
    | Node (y, l, r) ->
        if x < y
           then Node (y, insert x l, r)
           else if x > y then Node (y, l, insert x r)
           else Node (y, l, r);;
val insert : 'a -> 'a tree -> 'a tree = <fun>
# let tree = Node (5, Node (1, Leaf, Leaf),
                      Node (7, Leaf, Leaf));;
val tree : int tree = Node (5, Node (1, Leaf, Leaf),
                               Node (7, Leaf, Leaf))
# let tree' = insert 6 tree;;
val tree' : int tree =
  Node (5, Node (1, Leaf, Leaf),
           Node (7, Node (6, Leaf, Leaf), Leaf))
```

The Option Type

- Important builtin type
- Used to write partial functions

```
# type 'a option = Some of 'a | None;;
type 'a option = Some of 'a | None
```

Records

- Labeled collections of values with arbitrary types
- Record types must be declared

```
# type point = { point_x : int; point_y : int};;
type point = { point_x : int; point_y : int; }
```

- Label names must be globally unique
- Record construction:

```
# let p = { point_x = 5; point_y = 3 };;
val p : point = {point_x = 5; point_y = 3}
```

- Field selection:

```
# let move p1 p2 =
    { point_x = p1.point_x + p2.point_x;
      point_y = p1.point_y + p2.point_y };;
val move : point -> point -> point = <fun>
# move p p;;
- : point = {point_x = 10; point_y = 6}
```

IO

Some functions for doing I/O:

```
val print_string : string -> unit
val print_endline : string -> unit
val prerr_string : string -> unit
val prerr_endline : string -> unit
val read_line : unit -> string

val open_out : string -> out_channel
val output_string : out_channel -> string -> unit

val open_in : string -> in_channel
val input_line : in_channel -> string
```

Compilation

- Files with OCaml source code have the extension `.ml`
- Compiler `ocamlc`: produces portable bytecode
- Compiler `ocamlopt`: produces fast native code
- Compiled program executes definitions in order of their appearance in the source file(s)
- Functions in some other source file `foo.ml` must be qualified with the prefix `Foo.`
- If file `bar.ml` uses functions from `foo.ml`, then `bar.ml` must come after `foo.ml` on the commandline. No cycles are allowed!

Compilation Example

- File `fib.ml`:

```
let fib = ...
```

- File `main.ml`:

```
let _ =
  let _ = print_string "Input some number: " in
  let line = read_line () in
  let i = int_of_string line in
  let j = Fib.fib i in
    print_endline ("Result: " ^ string_of_int j)
```

- Compilation: `ocamlc -o fib fib.ml main.ml`
- Produces file `fib`:

```
$ ./fib 6
Input some number: 6
Result: 8
```

Resources

- OCaml Homepage: `http://caml.inria.fr/`
- Language Manual: `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`
- The standard library: `http://caml.inria.fr/pub/docs/manual-ocaml/libref/`
- Jason Hickey: Introduction to the Objective Caml Programming Language. (The slides are based on this script) `http://files.metaprl.org/doc/ocaml-book.pdf`
- Emmanuel Chailloux, Pascal Manoury and Bruno Pagano: Developing Applications with Objective Caml `http://caml.inria.fr/pub/docs/oreilly-book/html/index.html`