IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Functional programming

Introduction to OCaml

Fabien Dagnat
ELU 610 – C6
1st semester 2019

Plan

OCaml

- General purpose language developed by INRIA since 1990…
- … and now widely used by industrials (Airbus, ANSSI, CEA, Be-Sport, Bloomberg, Facebook, Jane Street Capital, Tezos, …)
- `http://ocaml.org`
- A book is downloadable at `https://realworldocaml.org`
- This lesson covers only the functional part (chap 1-7)
- Online simple tutorial at `http://try.ocamlpro.com`

OCaml industrial users

American Museum of Natural History
be sport®
Bloomberg

galois
Dassault Systemes
Jane Street
CiTRIX®

De la recherche à l'industrie cea
Agence nationale de la sécurité des systèmes d'information · ANSSI ·
docker

http://ocaml.org/learn/companies.html

Progress

λ-calcul $\subset$ OCaml

- λ-calcul
  - Variables are strings `beginning_by_lowercase_letter`
  - Application of `a` to `b` is just `a b`
  - (Anonymous) function `function x -> body`
- Some syntactic sugar
  - A function with several arguments `fun x y -> body`
    - equivalent to `function x -> function y -> body`
  - Naming a value `let x = value in body`
    - note that the scope of `x` is explicit (here `body`)
    - equivalent to `(function x -> body) value`
  - Naming a function `let f x y = value in body`
    - equivalent to `let f = fun x y -> value in body`
  - Sequencing `a ; b`
    - equivalent to `let _ = a in b`

Evaluation order

- Computation use call by value
- evaluate arguments before application
- ⚠ evaluation order is undefined between the arguments of a function
- computing `f a1 a2` computes
  1. `f`[1], `a1` and `a2` in a unspecified order
  2. computes the call
- if you need a specific order, use **`let`**:

```
let f = ... in
let a1 = ... in
let a2 = ... in
f a1 a2
```

---

[1] the function may be any expression

Toplevel expressions and execution

- A toplevel expression is an expr. not contained in a larger expr.
- A toplevel naming expression (**`let`**) without a scope (no **`in`**)
  - has its scope extended to all the following toplevel expr.
  - provides a kind of global naming
- There is two ways of executing OCaml programs
  1. using a(n interactive) REPL[2] (an interpreter), `utop` or `ocaml`
     it reads an expr., evaluates it and then prints the result
  2. using a compiler and then executing the produced executable file
- In a REPL, you enter an expr. and terminates it by `;;`
- The compilers consider that two expr. separated by a (blank) line are toplevel expr. in sequence

---

[2] Read-eval-print-loop

Practicing

- The identity function
- A function applying a function to a value
- A function composing two functions

Practicing

- The identity function

  `let id = function x -> x` or `let id x = x`

- A function applying a function to a value

- A function composing two functions

Practicing

- The identity function

  `let id = function x -> x` or `let id x = x`

- A function applying a function to a value

  `let eval = function f -> function x -> f x` or
  `let eval f = function x -> f x` or `let eval f x = f x`

- A function composing two functions

Practicing

- The identity function

  `let id = function x -> x` or `let id x = x`

- A function applying a function to a value

  `let eval = function f -> function x -> f x` or
  `let eval f = function x -> f x` or `let eval f x = f x`

- A function composing two functions

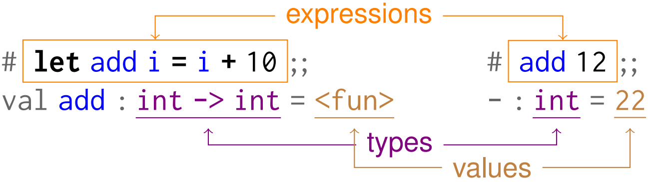
  `let compose f g x = f (g x)` or
  `let compose f g = function x -> f (g x)`
  Partial application consists in providing less arguments than the expected ones

Typing

- OCaml is a typed language with
  - primitive types
    - `bool`, `int`, `float`, `char`, `string`, ...
  - type constructors (build new types from existing types)
    - `t list` for lists of type `t`, `t1 * t2` for pairs of type `t1` and `t2`, ...
    - `t1 -> t2` for functions from `t1` to `t2`
- Typing is static: typing correctness is checked before execution
- Types of expressions are inferred (computed) by the compiler

⇒ The programmer is not required to give them!

- The REPL prints them for toplevel expr.



```
# let add i = i + 10;;             # add 12;;
val add : int -> int = <fun>       - : int = 22
```

Typing is strict

- Typing is strict, each expression must be correctly typed

```
# add "tutu" ;;
Error:          This expression has type string but an
expression was expected of type int
```

- There is no automatic conversion

```
# add 12.1 ;;
Error: This expression has type float but an expression
was expected of type int
# add (int_of_float 12.1) ;;
- : int = 22
```

- OCaml has no overloading: a name has only one type

```
# let pi = 4.0 * atan 1.0 ;;
Error: This expression has type float but an expression
was expected of type int
# let pi = 4.0 *. atan 1.0 ;;
val pi : float = 3.141592653589793
```

Polymorphism

- What is the type of the identity function? `let id x = x`

Polymorphism

- What is the type of the identity function? `let id x = x`
- `id` can take any value as argument and returns this value

⇒ Types may contain type variables `'a`, `'b`, ...

- The type of `id` is ∀`'a.'a -> 'a`
- This is called (universal) polymorphism

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

the universal quantification is implicit

- What is the type of the identity function? `let id x = x`
- `id` can take any value as argument and returns this value

⇒ Types may contain type variables `'a`, `'b`, ...

- The type of `id` is ∀`'a.'a -> 'a`
- This is called (universal) polymorphism

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

# let eval f x = f x ;;
val eval :        ?          = <fun>

# let compose f g x = f (g x) ;;
val compose :         ?                  = <fun>
```

Polymorphism

- What is the type of the identity function? **`let`** `id x = x`
- `id` can take any value as argument and returns this value

⇒ Types may contain type variables `'a`, `'b`, ...

- The type of `id` is ∀`'a.'a -> 'a`
- This is called (universal) polymorphism

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

# let eval f x = f x ;;
val eval : ('a -> 'b) -> 'a -> 'b = <fun>

# let compose f g x = f (g x) ;;
val compose :                  ?                        = <fun>
```

Polymorphism

- What is the type of the identity function? `let id x = x`
- `id` can take any value as argument and returns this value

⇒ Types may contain type variables `'a`, `'b`, ...

- The type of `id` is ∀`'a`.`'a -> 'a`
- This is called (universal) polymorphism

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

# let eval f x = f x ;;
val eval : ('a -> 'b) -> 'a -> 'b = <fun>

# let compose f g x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Primitive data types

- `()` the nothing value of type `unit`
- **`false`** and **`true`** of type `bool`
  - logical operators: **`not`**, **`&&`**, **`||`**, ...
  - comparison operators: **=**, **<>**, **<**, **>**, **<=**, **>=**
- integers of type `int`
  - usual operators: **`+`**,**`-`**,**`*`**, **`/`**, **`mod`**, `int_of_float`, ...
- floating number of type `float`
  - usual operators: **`+.`**,**`-.`**,**`*.`**, **`/.`**, **`**`**, `float_of_int`, ...
- `'a'`, `'\n'`, ... of type `char`
- `"\ta string\n"` of type `string`
  - concatenation by **`^`**
  - conversion of the primitive data types by `string_of_`*type*
  - char at position `i` by `str`.[`i`]

Type constructors: product

- Tuples (`e1,...,en`) of type `t1 * ... * tn`
  - no function to decompose (see later pattern matching)
  - pairs when $n = 2$, decompose using `fst` and `snd`
- If a vector is represented by a pair, compute its norm
- A function applying two functions to a pair

Type constructors: product

- Tuples (`e1,...,en`) of type `t1 * ... * tn`
  - no function to decompose (see later pattern matching)
  - pairs when $n = 2$, decompose using `fst` and `snd`
- If a vector is represented by a pair, compute its norm

```
# let square x = x *. x ;;
val square : float -> float = <fun>
# let norm c = sqrt (square(fst c) +. square(snd c)) ;;
val norm : float * float -> float = <fun>
# norm (2.0,-1.0) ;;
- : float = 2.23606797749979
```

- A function applying two functions to a pair

Type constructors: product

- Tuples (`e1,...,en`) of type `t1 * ... * tn`
  - no function to decompose (see later pattern matching)
  - pairs when $n = 2$, decompose using `fst` and `snd`
- If a vector is represented by a pair, compute its norm

```
# let square x = x *. x ;;
val square : float -> float = <fun>
# let norm c = sqrt (square(fst c) +. square(snd c)) ;;
val norm : float * float -> float = <fun>
# norm (2.0,-1.0) ;;
- : float = 2.23606797749979
```

- A function applying two functions to a pair

```
# let apply f g c = f (fst c), g (snd c) ;;
val apply : ('a -> 'b) -> ('c -> 'd) -> 'a * 'c -> 'b * 'd = <fun>
# apply (fun x -> x + 1) (fun x -> x - 1) (4,4) ;;
- : int * int = (5,3)
```

Patterns

- A pattern is an expression made of
  - value constructors and values
  - variables (only one occurrence for each variable)
  - holes: `_`

  `(1,true)`, `1`, `(1,_,x)` are patterns
- A pattern may match a value
  - if they have the same (constructor) structure
  - variables and holes match any value

  `(1,_,x)` matches `(1,"er",'a')` but neither `1` nor `(2,"er",'a')`
- When a pattern matches a value, its variables are bound to the corresponding parts of the value
  when `(1,_,x)` matches `(1,"er",'a')` it binds `x` to `'a'`
- The let syntax is **`let`** *pattern* = *expression* [**`in`** *expression*]

Pattern matching

- Main control structure, used to decompose values
- A pattern matching case is a pattern and an expr.
  - when "applied" to a value, it can succeed or fail
  - if it succeeds, expr. is evaluated with the variables bound
  - syntax: *pattern* **->** *expression*
- A pattern matching is a sequence of pattern matching cases
  - when "applied" to a value, it uses the first case to try to match
  - if it fails, the next case is used
  - and so on, until one case matches
  - if none of the cases matches, there is a `Match_failure` exception
  - **`function`** p_1 **->** e_1 **|** . . . **|** p_n **->** e_n[3]
  - **`match`** e **`with`** p_1 **->** e_1 **|** . . . **|** p_n **->** e_n
    - equivalent to (**`function`** p_1 **->** e_1 **|** . . . **|** p_n **->** e_n) e
- If a case is useless or is missing, the typer will raise a warning

---

[3] **`fun`** can only have one case

Practice of pattern matching

- Compute $a \Rightarrow b$ for a pair of boolean (a, b)

- A function testing if an integer is zero

Practice of pattern matching

- Compute $a \Rightarrow b$ for a pair of boolean (a, b)

```
# let imply v = match v with
  | (true ,true) -> true | (true ,false) -> false
  | (false,true) -> true | (false,false) -> true ;;
val imply : bool * bool -> bool = <fun>
```

or

```
# let imply = function
  | (true ,x) -> x
  | (false,_) -> true ;;
val imply : bool * bool -> bool = <fun>
```

- A function testing if an integer is zero

Practice of pattern matching

- Compute $a \Rightarrow b$ for a pair of boolean (a, b)

```
# let imply v = match v with
  | (true ,true) -> true | (true ,false) -> false
  | (false,true) -> true | (false,false) -> true ;;
val imply : bool * bool -> bool = <fun>
```

or

```
# let imply = function
  | (true ,x) -> x
  | (false,_) -> true ;;
val imply : bool * bool -> bool = <fun>
```

- A function testing if an integer is zero

```
# let is_zero = function
  | 0 -> true
  | _ -> false ;;
val is_zero : int -> bool = <fun>
```

Lists

- Lists are built from
  - the empty list: []
  - an element e and a list l: e`::`l
    - e is the head of e`::`l
    - l is the tail of e`::`l
- Lists are monomorphic
  - all elements in a list must have the same type
  - a list of elements of type `t` is of type `t list`
- Syntactic sugar: $[e_1$`;`$\ldots$`;`$e_n]$ is equivalent to e_1`::`$\ldots$`::`e_n`::`[]
- Utility functions for lists
  - concatenation **`@`**
  - lots of utility functions in library `List` (head by `hd`, tail by `tl`, ...)

Recursive functions

- In functional languages, iteration is done by recursion
- Recursive functions are defined by **`let rec`**
- Recursion is often combined with pattern matching

```
# let rec insert elt = function
  | [] -> [elt]
  | h::t when elt <= h -> elt::h::t
  | h::t -> h::(insert elt t) ;;
val insert : 'a -> 'a list -> 'a list = <fun>
```

- Computing Fibonacci numbers

Recursive functions

- In functional languages, iteration is done by recursion
- Recursive functions are defined by **`let rec`**
- Recursion is often combined with pattern matching

```
# let rec insert elt = function
  | [] -> [elt]
  | h::t when elt <= h -> elt::h::t
  | h::t -> h::(insert elt t) ;;
val insert : 'a -> 'a list -> 'a list = <fun>
```

- Computing Fibonacci numbers

```
# let rec fib = function
  | n when n < 0 -> failwith "error"
  | 0 -> 0
  | 1 -> 1
  | n -> fib (n-1) + fib (n-2) ;;
val fib : int -> int = <fun>
```

Functions are values

- Functions can
  - take functions as arguments
  - return functions
  - can be applied partially
- For example, applying a function on all elements of a list

```
# let rec iter f = function
  | [] -> ()
  | h::t -> f h; iter f t ;;
 val iter : ('a -> 'b) -> 'a list -> unit = <fun>
#  iter print_string ["a";"b";"c";"n"] ;;
abcn
- : unit = ()
#  let print_list = iter print_string ;;
val print_list : string list -> unit = <fun>
```

Progress

Another product type constructors: records

- Records: product types naming the sub-elements

```
# type ratio = { num : int ; den : int } ;;
type ratio = { num : int; den : int; }
```

- A value of type `ratio` can be defined by

```
# let r1 = { num = 1 ; den = 16 } ;;
val r1 : ratio = {num = 1; den = 16}
# let r2 = { r1 with num = 3 } ;;
val r2 : ratio = {num = 3; den = 16}
```

  the order in which the fields are given is unimportant

- The field value are accessed by their name

```
# let add r1 r2 = {
  num = r1.num * r2.den + r2.num * r1.den ;
  den = r1.den * r2.den } ;;
val add : ratio -> ratio -> ratio = <fun>
```

Sum types

- Enumerations

```
# type dir = North | South | East | West ;;
type dir = North | South | East | West
```

- Generalized by variants

```
# type number = Int of int | Float of float | Error ;;
type number = Int of int | Float of float | Error
```

- `Int` 8, `Float` `5.4` and `Error` are of type `number`

```
# (Int 8, Float 5.4, Error) ;;
- : number * number * number = (Int 8, Float 5.4, Error)
```

- Values of variant types are manipulated by pattern matching

```
# let print_number = function
  | Int n -> print_int n
  | Float f -> print_float f
  | Error -> print_string "error" ;;
val print_number : number -> unit = <fun>
```

Sum types II

- Sum types can be parameterized

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

- Sum types can be recursive

```
# type 'a list = [] | :: of 'a * 'a list ;;
type 'a list = [] | :: of ’a * ’a list
```

- Both the option and list types are already defined in OCaml

⚠ In fact in OCaml, the variant constructors must be

1. a capitalized identifier
2. []
3. `::`[4], it will be treated as a binary infix constructor

---

[4] to be correct the type declaration should surrounds it by parenthesis

Progress

Module

- A module is a set of type, value and function definitions[5]
- A module has
    - a signature defining its public interface
        - type definitions and type declarations for values and functions
    - a structure defining its content
        - any OCaml code
- Elements of the signature must be part of the structure
- In another module, one can use a **public** element `elt` of a module `M`
    - by `M.elt`
    - by `elt` if the module was previously opened by **open** `M`
- A `filename.ml` file is a module `Filename`
    - if there is a `filename.mli`, it provides its signature
    - else everything is public (which is a bad practice)

---

[5] and modules but we won't cover that

Abstraction

- One of the interest of modules is abstracting (hiding) types
- The following signature abstracts `ratio`

```
type ratio
val create : int -> int -> ratio                 A constructor
val add : ratio -> ratio -> ratio                A manipulator
val print : ratio -> unit                        A destructor
```

- Code outside the defining module of an abstract type
  - cannot use its implementation
  - can only manipulate value through the offered functions

⇒ Changing the implementation of `ratio` does not impact clients

- Abstracting internal functions and values is also a good idea
- The signature is the ApplicationProgrammingInterface
  - it generally includes constructors, manipulators and destructors for the abstract types

Exceptions

- Exceptions are declared by **exception**

```
# exception Empty_list of string ;;
exception Empty_list of string
```

- Raised by `raise`

```
# let head = function
  | [] -> raise (Empty_list "bouh!")
  | hd :: tl -> hd ;;
val head : 'a list -> 'a = <fun>
```

- Caught by **try with**

```
try
  head []
with
  | Empty_list msg -> print_endline msg ;;
bouh!
- : unit = ()
```

Progress

1. OCaml basics
2. More type constructors
3. Modules
4. **Executing and Building**
5. Conclusion

Compiling and running OCaml

- `ocaml` REPL (we use `utop`)
  - compiles and executes immediately
  - prints value and types
  - provides a simple line editor (bash default binding)
  - `#use "toto.ml";;` loads and execute every expr. of `toto.ml`
- Two compilers exist
  - a bytecode compiler `ocamlc` (with bytecode interpreter `ocamlrun`[6])
  - a native compiler `ocamlopt` that directly produces executable files
- Both are three steps compilers (`XX` means `c` or `opt`)
  - compile signatures by `ocamlXX -c YY.mli` to produce `YY.cmi`
  - compile modules by `ocamlXX -c YY.ml` to produce `YY.cmZZ`
    - `ZZ` = `o` if `XX` = `c` and `x` if `XX` = `opt`
  - linking of all the need modules by `ocamlc -o WW YY1.cmZZ YY2.cmZZ` to produce the executable `WW`
- For a module without signature all its content is put in the `cmi`

---

[6] Invoking `ocamlrun` is optional since the bytecode file already invoke it

Automatic building `ocamlbuild`

- All files are compiled in the directory `_build`
- It groans if it finds compilation artefacts elsewhere!
- It has a target `X.byte` or `X.native` to indicate the compiler
  - `ocamlbuild -libs unix main.native`
  - `X` become the name of the executable
- Finds all the dependencies (hence the compilation order) alone
- Can run if you add `--` followed by the command line args.
  `ocamlbuild main.byte -- file.txt`
- Configuration file `_tags` for a finer control of build
- `https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc`

Progress

Conclusion

- Short introduction to the functional core of OCaml
  « Computing values not modifying variables »
- It is our objective during this module
- We ignore
  - imperative features
  - objects
  - first class modules, ...
- You need to practice...
- `http://ocaml.org`
- `https://realworldocaml.org` (chap 1-7 and 16)