Ride the Caml
An introduction to Caml

Damien Martin-Guillerez
SED
CENTRE Inria
BORDEAUX SUD-OUEST

02 OCT. 2012

INTRODUCTION

Caml was developed at Inria since 1985.

- **functional** and imperative programming

- **Strong typing** and **type-based pattern matching**

- **Polymorphic types** (a.k.a templates)

- Inner list type and easy construction of complex types

Some success stories...

Caml is beautiful and has found its bride many times:

- Education: **CPGE** computer science courses

- Research & Community: **Coq**, **Ocsigen**

- Industry: **Microsoft F#**, ASTRÉE (**Airbus**), Financial computing (**Jane Street Capital**)

- Free software: **MLDonkey**, **MediaWiki** mathematical formulas

OUTLINE

1. The basics
2. Functional programming
3. Imperative programming
4. Type-based pattern matching
5. More fun!

1

The basics

Hello, World!

```
print_string "Hello, World!\n";;
```

You said types?

Everything has a type!

- Basic types: `int`, `float`, `string`, `char`, `bool`, `unit` ...
  - An `int` is not a `float`!!!
  - The `unit` type has only one value: `()` (a.k.a void)

- Inner vector, list and tuple types
  - `'a vect: [| |]; int vect: [| 1 |] - [|1; 2 |]`
  - `'a list: []; int list: [1] - [1; 2]`
  - `int * int: 1, 2; int * float: 1, 2.0`

- Caml is functional, so function type:

```
int * int -> int, int -> int -> int
```

- Caml has also complex types similar to structures and enums...

Erk! Types!

```
#let x = 1
and y = 1.;;
x : int = 1
y : float = 1.0

#x + 2;;
- : int = 3

#y + 2;;
Toplevel input:
>y + 2;;
>^
This expression has type float, but is used with type int.

#y +. 2;;
Toplevel input:
>y +. 2;;
>     ^
This expression has type int, but is used with type float.

#y +. 2.;;
- : float = 3.0

#x + 2.;;
Toplevel input:
>x + 2.;;
>    ^^
This expression has type float, but is used with type int.
```

Yeah types!

- Polymorphism (a.k.a template)

```
#[];;
- : 'a list = []
#let do_something_else x y = 1 ;;
do_something_else : 'a -> 'b -> int = <fun>
```

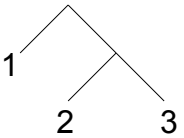
- Type inference

```
#let get_something a = a+a;;
get_something : int -> int = <fun>
#let do_something x =
    let y = get_something x in do_something_else y x;;
do_something : int -> int = <fun>
```

Yeah more types!

- Complex types

```
#type 'a BinTree =
     Leaf of 'a
   | Node of ('a BinTree * 'a BinTree)
;;
Type BinTree defined.
#Node (Leaf 1, Node (Leaf 2, Leaf 3));;
- : int BinTree = Node (Leaf 1, Node (Leaf 2, Leaf 3))
```



- Record types

```
#type 'a record = { name: string; object: 'a };;
Type record defined.
#{ name = "name"; object = "value"};;
- : string record = {name = "name"; object = "value"}
```

Some operators (1/2)

- On integers

```
1 + 2;;        1 - 2;;        1 * 2;;        1 / 2;;        1 mod 2;;
- : int = 3    - : int = -1   - : int = 2    - : int = 0    - : int = 1
```

- On floating points

```
1. +. 2.;;       1. -. 2.;;       1. *. 2.;;       1. /. 2.;;
- : float = 3.0  - : float = -1.0  - : float = 2.0  - : float = 0.5
```

- On strings

```
"a" ^ "b";;        sub_string "abc" 0 1;;   string_length "abc";;
- : string = "ab"  - : string = "a"          - : int = 3
```

- On lists

```
let l = [1; 2; 3];;
l : int list = [1; 2; 3]
hd l;;
 - : int = 1
1::l;;
- : int list = [1; 1; 2; 3]
```

```
list_length l;;
- : int = 3
tl l;;
- : int list = [2; 3]
1.::l;;
```

Some operators (2/2)

- On vectors

```
let v = [| 1; 2; 3 |];;          vect_length v;;    sub_vect v 0 1;;
v : int vect = [|1; 2; 3|]       - : int = 3        - : int vect = [|1|]
v.(0);;          v.(0) <- 1;;    v.(0);;            v <- 2;;
- : int = 1      - : unit = ()   - : int = 1
```

- On references

```
let r = ref 1;;          !r;;           r := 2;;          !r;;
r : int ref = ref 1      - : int = 1    - : unit = ()     - : int = 2
r := [| 1 |];;           r <- 2;;
```

- On records

```
type 'a record = { name: string; mutable object: 'a };;
Type record defined.
let r = {name = "name"; object = 1};;
r : int record = {name = "name"; object = 1}
r.name;;                 r.object;;     r.object <- 2;;   r.object;;
- : string = "name"      - : int = 1    - : unit = ()     - : int = 2
                                        r.name <- 1;;
```

Some basic functions

- Types so type conversion
  - `string_of_int : int → string`
  - `string_of_float : float → string`
  - `float_of_int : int → float`
  - `int_of_float : float → int`
  - `int_of_string : string → int`
  - ...
- Functions with side-effects
  - `print_string : string → unit`
  - `print_int : int → unit`
  - ...

2

Functional programming

Functional?

- Functions are objects
  - Typed

```
let f x y = x + y;;
f : int -> int -> int = <fun>
```

```
f 1 2;;
- : int = 3
```

  - Value that can be passed as argument

```
let f2 f y = f 1 y;;
f2 : (int -> 'a -> 'b) -> 'a -> 'b = <fun>
```

```
f2 f 1;;
- : int = 2
```

  - Lambda / anonymous functions

```
(fun x y -> x + y);;
- : int -> int -> int = <fun>
```

```
(fun x y -> x + y) 1 2;;
- : int = 3
```

  - Curryfication

```
let f3 (x,y) = x + y;;
f3 : int * int -> int = <fun>
```

```
f3 (1,2);;
- : int = 3
```

```
let f4 = f 1;;
f4 : int -> int = <fun>
```

```
f4 2;;
- : int = 3
```

Without recursivity, we are nothing.

- The `rec` keyword enables recursivity on a function

```
let rec length l =                    length : 'a list -> int = <fun>
  if l = []
    then 0
    else 1 + length (tl l)
;;

length l;;                            - : int = 3
```

- Prefer tail-recursive functions!

```
let length l =                        length : 'a list -> int = <fun>
  let rec aux n l1 =
    if l1 = []
      then n
      else aux (n+1) (tl l1)
  in
    aux 0 l
;;

length l;;                            - : int = 3
```

3

Imperative programming

Berk!

```
let length l =                                  length : 'a list -> int = <fun>
  let it = ref l
  and r = ref 0
  in
    while !it <> [] do
      r := !r + 1;
      it := tl !it
    done;
    !r
;;
length [1,2,3];;                                - : int = 1


let sum v =                                     sum : int vect -> int = <fun>
  let s = ref 0
  in
    for i = 0 to (vect_length v) - 1 do
      s := !s + v.(i)
    done;
    !s
;;
sum [| 1; 2; 3 |];;                             - : int = 6
```

4

Type-based pattern matching

The match operator

```
let length l =
   let rec aux r l1 =
      match l1 with
         | []     -> r
         | (_::q) -> aux (r+1) q
      in
         aux 0 l
;;
length : 'a list -> int = <fun>

length [1,2,3];;
- : int = 1
```

Matching with more fun!

```
let length l =
   let rec aux r = fun
         | []     -> r
         | (_::q) -> aux (r+1) q
      in
         aux 0 l
;;
length : 'a list -> int = <fun>

length [1,2,3];;
- : int = 1
```

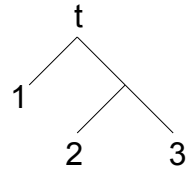
The power of type-based pattern matching Step 1

```
let rec map f = fun
   | []     -> []
   | (t::q) -> (f t) :: (map f q)
;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

map (fun x -> x + 1) [1; 2; 3];;
- : int list = [2; 3; 4]
```

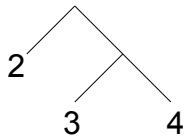
The power of type-based pattern matching Step 2

```
type 'a BinTree = Leaf of 'a | Node of ('a BinTree * 'a BinTree);;
Type BinTree defined.
#let t = Node (Leaf 1, Node (Leaf 2, Leaf 3));;
t : int BinTree = Node (Leaf 1, Node (Leaf 2, Leaf 3))
```




```
let rec bt_map f = fun
  | (Leaf x)        -> Leaf (f x)
  | (Node (t1, t2)) -> Node ((bt_map f t1),(bt_map f t2))
;;
bt_map : ('a -> 'b) -> 'a BinTree -> 'b BinTree = <fun>

bt_map (fun x -> x + 1) t;;
- : int BinTree = Node (Leaf 2, Node (Leaf 3, Leaf 4))
```




```
let rec bt_nb_leaf = fun
  | (Leaf _)        -> 1
  | (Node (t1, t2)) -> (bt_nb_leaf t1) + (bt_nb_leaf t2)
;;
bt_nb_leaf : 'a BinTree -> int = <fun>

bt_nb_leaf t;;
- : int = 3
```

The power of type-based pattern matching Step 3

```
type 'a record = { name: string; object: 'a };;
Type record defined.

let rec record_apply f = fun
  | []                                  -> []
  | ({ name = n; object = o }::q) -> { name = n; object = (f o) }::
                                        (record_apply f q)
;;
record_apply : ('a -> 'b) -> 'a record list -> 'b record list =
<fun>

record_apply (fun x -> x + 1)
             [ {name = "1"; object = 1}; {name = "2"; object = 2}];;
- : int record list =
             [{name = "1"; object = 2}; {name = "2"; object = 3}]
```

And the icing on the cake...

It is fast!

5

More fun!

Strong typing...

- … is good for health
  - Detects errors
  - Type specification
  - Type-based proof
- … is not problematic thanks to
  - Type inference
  - Pattern-matching

Caml is good for your health!

- Pattern-matching on types
- Functional and imperative
- Strong-typing and type inference
- Good performance
  - Native compiler
  - Optimized pattern matching
  - Optimized recursion especially for tail-recursivity

Try it!

http://caml.inria.fr