Initiation au langage Objective Caml

David Monniaux

http://www.di.ens.fr/~monniaux

David.Monniaux@polytechnique.edu

Organisation du cours

Questions Enseignants de TD + David.Monniaux@polytechnique.edu

http://www.dix.polytechnique.fr/profs/informatique/David.Monnia
maj1_caml_2005/

Généralités

Présentation rapide

Objective Caml est un langage fonctionnel, fortement typé, non spécialisé.

Successeur de Caml Light (lui même successeur du « Caml lourd »). De la famille ML. Autre membre connu : SML (« Standard ML »).

Conçu et implémenté à l'INRIA-Rocquencourt par Xavier Leroy et d'autres.

Utilisations principale : enseignement, calcul symbolique, logiciels d'analyse statique (Microsoft, ENS), mais aussi logiciels de manipulation de fichiers (Unison, MLDonkey...)

Typage fort

En Caml, les variables ont un type. Pas de passage « mystérieux » d'un type à l'autre. Pas de jeu sur les pointeurs comme en C. Types référence mais sans « valeur nulle ».

Conséquence sécurité d'exécution : impossible d'avoir des « segmentation fault » ou « null pointer exception ».

Contrepartie parfois plus lourd que des langages non typés ou à typage faible comme Scheme, Common Lisp ou Perl

Compilation et interprétation

Toplevel `ocaml`

```
        Objective Caml version 3.08.2

#
```

Compilation bytecode

```
ocamlc foobar.ml -o foobar
```

Compilation native

```
ocamlopt foobar.ml -o foobar
```

Évaluation

Expressions

presque tout dans le langage est une expression

```
        Objective Caml version 3.08.2

# 5;;
- : int = 5
```

;; = terminateur de phrase pour le toplevel (en général facultatif dans code compilé)

Arithmétique

Entiers

```
# 5 + 5;;
- : int = 10
```

Flottants

```
# 6. +. 7.;;
- : float = 13.
```

Remarquer +.

x

Maths

Fonctions (voir documentation)

```
# sin 3.;;
- : float = 0.14112000805986721
# 2. +. (sin 3.);;
- : float = 2.1411200080598672
```

Attention

Il n'y a jamais de conversion implicite, même pas `int` $\longrightarrow$ `float`.

```
# 2 +. 3.;;
Characters 0-1:
  2 +. 3.;;
  ^
This expression has type int but is here used with type float
# (float_of_int 2) +. 3.;;
- : float = 5.
```

Let...

```
# let x = 4 in x + 2;;
- : int = 6
```

Valide pour toutes les expressions suivantes :

```
# let x = 4;;
val x : int = 4
# x + 2;;
- : int = 6
```

Tests

```
# 5 > 4;;
- : bool = true
# if 5>4 then 2 else 3;;
- : int = 2
# if 5>4 then 2 else "3";;
Characters 19-22:
  if 5>4 then 2 else "3";;
                     ^^^
This expression has type string but is here used with type int
```

Produits

```
# (4, 5);;
- : int * int = (4, 5)
# (2, "xyz", 3);;
- : int * string * int = (2, "xyz", 3)
# let x = (2, 3) in fst x;;
- : int = 2
```

Types fonctionnels

Fonctions : motivation

En Java : fonction = objet qu'on peut exécuter
= objet avec une méthode que l'on peut exécuter.

```
interface FunctionIntString {
    String eval(int x);
}
```

En Caml :

```
# fun x -> (string_of_int x);;
- : int -> string = <fun>
```

Fonctions : utilisation en Java

```
abstract class FunctionIntString {
    abstract String eval(int x);

    String[] mapArray(int [] in) {
        String[] out = new String[in.length];
        for(int i=0; i<in.length; i++) {
            out[i] = eval(in[i]);
        }
        return out;
    }
}
```

Fonctions : utilisation en Java (2)

```
class IntToStringSurround extends FunctionIntString {
    String prefix, postfix;

    IntToStringSurround(String aPrefix, String aPostfix) {
        prefix = aPrefix;
        postfix = aPostfix;
    }

    public String eval(int x) {
        return (prefix + x + postfix);
    }
```

Fonctions : utilisation en Java vs Caml

```
public static void main(String[] args) {
    IntToStringSurround obj = new IntToStringSurround("<", ">");
    String[] t = obj.mapArray(new int[] {1, 3, 5});
}
```

Caml :

```
Array.map string_of_int [1; 3; 5];
```

Définition et utilisation

Écriture de base

```
# fun a -> a+1;;
- : int -> int = <fun>
```

Abréviation

```
# let f = fun a -> a+1;;
val f : int -> int = <fun>
```

Plus court :

```
# let f2 a = a+1;;
val f2 : int -> int = <fun>
```

Plusieurs arguments

Via types produits

```
# let f (a, b) = a+b;;
val f : int * int -> int = <fun>
```

Curryfication (Haskell Curry)

$A \times B \to C \simeq A \to (B \to C)$ noté $A \to B \to C$

```
# let f2 a b = a+b;;
val f2 : int -> int -> int = <fun>
```

À retenir Style préféré en Caml : currifié (favorisé par les optimisations du compilateur).

Polymorphisme

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

Valable pour tout 'a (α), \'b (β), \'c

Récurrence

$n \mapsto n! + 3$

```
# let f n = let rec fact x =
  if x > 0
  then x * (fact (x - 1))
  else 1 in (fact n) + 3;;
val f : int -> int = <fun>
# f 5;;
- : int = 123
```

Types sommes

Types sommes simples

Valeurs « à choix multiple ».

```
# type texte_legal = DECRET | ARRETE | LOI | ORDONNANCE;;
type texte_legal = DECRET | ARRETE | LOI | ORDONNANCE
```

Filtrage

```
# let est_reglementaire texte =
    match texte with
      DECRET -> true | ARRETE -> true
    | LOI -> false | ORDONNANCE -> false;;
val est_reglementaire : texte_legal -> bool = <fun>
```

```
# est_reglementaire DECRET;;
- : bool = true
```

Filtrage (2)

```
# let est_reglementaire texte =
    match texte with
      (DECRET | ARRETE) -> true
    | (LOI | ORDONNANCE) -> false;;
val est_reglementaire : texte_legal -> bool = <fun>
# est_reglementaire DECRET;;
- : bool = true
```

Types sommes

```
# type donnee = String of string | Int of int;;
type donnee = String of string | Int of int
# let string_of_donnee x =
  match x with
    String s -> s
  | Int i -> string_of_int i;;
```

Types inductifs

```
type op_binaire = PLUS | MOINS | MULTIPLIER;;

type op_unaire = NEGATION | INVERSION_BINAIRE;;

type calcul_entier = Constante of int
  | Binaire of op_binaire * calcul_entier * calcul_entier
  | Unaire of op_unaire * calcul_entier;;
```

Récurrence sur types inductifs

```
# let rec evaluer = function
    Constante x -> x
  | Binaire(op, e1, e2) -> let v1=evaluer e1 and v2=evaluer e2
    (match op with
     PLUS -> v1+v2
     | MOINS -> v1-v2
     | MULTIPLIER -> v1*v2)
  | Unaire(op, e) -> let v = evaluer e in
    (match op with
     NEGATION -> -v
    | INVERSION_BINAIRE -> lnot v);;
val evaluer : calcul_entier -> int = <fun>
```

Motifs imbriqués

```
# let rec evaluer = function
    Constante x -> x
  | Binaire(PLUS, e1, e2) -> (evaluer e1) + (evaluer e2)
  | Binaire(MOINS, e1, e2) -> (evaluer e1) + (evaluer e2)
  | Binaire(MULTIPLIER, e1, e2) -> (evaluer e1) + (evaluer e2)
  | Unaire(NEGATION, e) -> - (evaluer e)
  | Unaire(INVERSION_BINAIRE, e) -> lnot (evaluer e);;
val evaluer : calcul_entier -> int = <fun>
# evaluer (Binaire(PLUS, Constante(4), Constante(5)));;
- : int = 9
```

Listes

Type prédéfini

```
# [1; 5; 6];;
- : int list = [1; 5; 6]
# 3 :: [1; 5; 6];;
- : int list = [3; 1; 5; 6]
# match [1; 5; 6] with x :: _ -> x;;
Characters 0-32:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
```

```
[]
  match [1; 5; 6] with x :: _ -> x;;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
- : int = 1
```

Enregistrements

```
# type fiche_eleve = {
  nom : string;
  prenom : string;
  matricule : int };;
# let fiche = { prenom = "Jean"; nom = "Dupont"; matricule = 45
val fiche : fiche_eleve = {nom = "Dupont"; prenom = "Jean"; mat
# { fiche with matricule = 457 };;
- : fiche_eleve = {nom = "Dupont"; prenom = "Jean"; matricule =
# fiche.prenom;;
- : string = "Jean"
```

Polymorphisme

```
type 'a arbre = Feuille of 'a
  | Noeud of ('a arbre) * ('a arbre);;
let rec appliquer_aux_feuilles f = function
  Feuille x -> Feuille (f x)
| Noeud(l, r) -> Noeud((appliquer_aux_feuilles f l),
                       (appliquer_aux_feuilles f r))

# Feuille 3;;
- : int arbre = Feuille 3
# appliquer_aux_feuilles string_of_int (Feuille 3);;
- : string arbre = Feuille "3"
```

Fonctionnalités impératives

Limites du fonctionnel pur

Jusqu'à présent Tout est une expression avec une valeur donnée.
Ordre d'évaluation indifférent.
Pas de variables.

Mais... Comment fait-on un affichage ?
Peut-on avoir de vraies variables ?

Affichage

```
# print_string "affiche";;
affiche- : unit = ()
# print_endline "affiche";;
affiche
- : unit = ()
# print_int 5;;
5- : unit = ()
# Printf.printf "<%d> %s\n" 4 "blabla";;
<4> blabla
- : unit = ()
```

Séquences

`()` unique élément du type `unit` ($\simeq$ `void` de C / Java ; `unit` parce que $T \times \texttt{unit} \simeq T$)

```
# 4 ; 5;;
Characters 0-1:
Warning: this expression should have type unit.
  4 ; 5;;
  ^

- : int = 5
```

A ; B exécute en séquence A puis B

Tests

```
# if 5 > 4 then print_endline "ok";;
ok
- : unit = ()
# if 5 > 4 then 6;;
Characters 14-15:
  if 5 > 4 then 6;;
                ^
This expression has type int but is here used with type unit
```

Tests...

```
# if 5 > 4 then (6; ());;
Characters 15-16:
Warning: this expression should have type unit.
  if 5 > 4 then (6; ());;
```

Attention à la syntaxe !

```
# if 5 > 4 then 6; ();;
Characters 14-15:
  if 5 > 4 then 6; ();;
                ^
This expression has type int but is here used with type unit
```

Boucles `for`

```
# for i = 1 to 5
  do
    Printf.printf "ligne <%d>\n" i
  done;;
ligne <1>
ligne <2>
ligne <3>
ligne <4>
ligne <5>
- : unit = ()
```

Références

Équivalent des variables.

```
# let x = ref 5;;
val x : int ref = {contents = 5}
# !x;;
- : int = 5
# x := 4;;
- : unit = ()
# !x;;
- : int = 4
```

Références...

Attention au ! à chaque utilisation !

```
# type 'a ref2 = { mutable contents2 : 'a };;
type 'a ref2 = { mutable contents2 : 'a; }
# let assign r x = r.contents2 <- x;;
val assign : 'a ref2 -> 'a -> unit = <fun>
# let ref2 x = { contents2 = x };;
val ref2 : 'a -> 'a ref2 = <fun>
```

Boucles

```
let rec fact n =
  let i = ref 1 and x = ref 1 in
  while !i <= n do
    x := !x * !i;
    incr i
  done;
  !x;;
```

Tableaux

```
# let t = [| 1; 3; 5 |];;
val t : int array = [|1; 3; 5|]
# Array.get t 1;;
- : int = 3
# Array.set t 0 5;;
- : unit = ()
# t;;
- : int array = [|5; 3; 5|]
```

Tableaux...

```
# let t = [| 1; 3; 5 |];;
val t : int array = [|1; 3; 5|]
# t.(1);;
- : int = 3
# t.(0) <- 5;;
- : unit = ()
# t;;
- : int array = [|5; 3; 5|]
```

Autres choses

Modules

Dans `a.ml` : fonction `f`.
Dans `b.ml` : appeler `A.f`.

```
ocamlc -o foobar a.ml b.ml
```

ou

```
ocamlc -c a.ml
ocamlc -c b.ml
ocamlc -o foobar a.cmo b.cmo
```

Modules paramétriques

```
module IntSet = Set.Make(struct
  type t = int
  let compare = ( - ) end);;

# IntSet.mem 4 (IntSet.singleton 5);;
- : bool = false
# IntSet.mem 4 (IntSet.singleton 4);;
- : bool = true
```

Système objets

Objective Caml permet de définir des classes avec de l'héritage multiple etc.

Système plus souple que celui de Java.

Recommandations

Consultez la documentation en ligne sur `http://caml.inria.fr`

Demandez conseil aux enseignants