Introduction à OCaml

Thomas Blanc

Terminale de l'école Alsacienne

4 octobre 2012

Robin Milner, ML et OCaml

Quelques dates

- 1973 : Robin Milner mets au point ML. Ancêtre d'OCaml à la University of Edinburgh.
- 1987 : L'INRIA publie Caml, qui reprend les principes de ML.
- 1996 : Xavier Leroy, Didier Rémy et Jérôme Vouillon mettent au point Objective Caml, qui sera renomé en OCaml.
- 2012 : L'INRIA publie OCaml 4.0, le langage est connu est utilisé dans de nombreux domaines.

Principes généraux

En quoi OCaml est spécial

Principes généraux

En quoi OCaml est spécial

- Un typage statique fort. Déterminé par inférence.

Principes généraux

En quoi OCaml est spécial

- Un typage statique fort. Déterminé par inférence.
- Multiparadigme : fonctionnel à effet de bords, objets, modules.

Principes généraux

En quoi OCaml est spécial

- Un typage statique fort. Déterminé par inférence.
- Multiparadigme : fonctionnel à effet de bords, objets, modules.
- Un outil extrémement puissant : le filtrage de motifs (ou pattern matching).

Principes généraux

En quoi OCaml est spécial

- Un typage statique fort. Déterminé par inférence.
- Multiparadigme : fonctionnel à effet de bords, objets, modules.
- Un outil extrémement puissant : le filtrage de motifs (ou pattern matching).
- Un excellent moyen de gérer les erreurs : le système d'exceptions.

Voyons quelques types...

Les valeurs de base :

- Les entiers, les flottants, les chaînes de caractères, les booléens...

Voyons quelques types...

Les valeurs de base :

- Les entiers, les flottants, les chaînes de caractères, les booléens...
- ...les listes, les tableaux...

Voyons quelques types...

Les valeurs de base :

- Les entiers, les flottants, les chaînes de caractères, les booléens...
- ...les listes, les tableaux...
- ...les types produits et somme...

Voyons quelques types...

Les valeurs de base :

- Les entiers, les flottants, les chaînes de caractères, les booléens...
- ...les listes, les tableaux...
- ...les types produits et somme...
- et les fonctions !

let !

Utiliser let

- let x = 1 ; ;
- let y = "bonjour" ; ;
- let f x = x + 1 ; ;

Attention à la portée des valeurs !

Le x qui sert de paramètre à f n'est pas le même que celui défini au début. Ils n'ont rien à voir !

La portée des valeurs

Portée des valeurs déclarées

Les variables déclarées avec "let x = ... ; ;" sont toujours accessible, sauf si une autre valeur du même nom est déclarée plus tard.
On parle de variable globale.

Portée des arguments d'une fonction

Les arguments d'une fonction ne sont accessible que depuis l'intérieur d'une fonction.
Si une valeur du même nom qu'un argument a déjà été déclarée, elle ne sera pas accessible depuis la fonction.
On parle de variable locale.

Déclarer des variables locales

O

n peut déclarer une variable locale au milieu d'un calcul avec let ... in :

- let x = 4 ; ;
- let y =
- let x2 = x*x in
- (x2-x)/2 ; ;

La portée de la variable x2 est limitée au calcul pour lequel elle a été déclarée "(x2-x)/2"

Faire des tests avec if ... then ... else ...

L'instruction if

Il est possible d'effectuer des tests avec l'instruction if :

- let f x =
- if x = 0
- then "zero"
- else "non-nul"

Il est très important de noter que les expressions de then et de else doivent être de même type !

La récursivité

Faire une boucle

On peut réutiliser le nom de la fonction qu'on déclare en ajoutant le mot clef rec :

- let rec factorielle n =
- if n <= 0
- then 1
- else n * (factorielle (n-1)) ; ;

Il ne faut pas oublier de mettre une condition pour arrêter la fonction, sinon elle ne s'arrête pas.

La récursivité terminale

Optimiser les appels récursifs

Une fonction récursive peut être optimisée par le compilateur si les appels récursifs sont toujours appelés en dernier.

- let factorielle n =
- let rec aux n res =
- if n <= 0
- then res
- else aux (n-1) (res*n)
- in
- aux n 1 ; ;

Les types

Déclarer un alias

On peut déclarer un alias tout simplement avec l'instruction type.

```
type entier = int ;;
```

Déclarer des types somme

On peut déclarer un type qui sera la somme d'autres types :

```
type nombre = Entier of entier | Reel of float | Rationnel of (entier * entier) | NaN ;;
```

Les types record

Déclarer un record

On peut déclarer un type somme dont les éléments sont associés à un nom :

```
type point = { x : int ; y : int } ; ;
```

Utiliser un record

```
let p = { x = 0 ; y = 15} ; ;
p.x ; ;
```

Les types paramétrés

Créer un type paramétré

```
type 'a arbre = Noeud of 'a arbre * 'a * 'a arbre | Feuille ; ;
```

Notez qu'une déclaration de type est toujours récursive.

Reconnaitre un type somme

Entrer dans un type somme se fait avec un filtrage

```
let x = Rationnel (5,6) ; ;
match x with
Rationnel (p,q) -> (float p) /. (float q)
| Entier n -> float n
| Reel f -> f
| NaN -> nan ; ;
```

Le piège du pattern matching

Attention à la portée des variables !

On ne peut pas comparer deux variables en pattern matching.
Pourquoi ?

La référence, la valeur modifiable

Déclarer et utiliser une référence

```
let x = ref 0 ; ;
x := 1 ; ;
x := !x + 4 ; ;
!x ; ;
```

Comportement

Si une référence est modifiée, même dans une fonction, sa valeur est modifiée partout. En fait, une référence n'est pas une valeur mais l'adresse d'une case mémoire où est stockée la valeur.
On peut aussi que modifier une référence est une action de type unit. Cela veut dire qu'elle ne renvoie aucune information.

La boucle for

Pour i allant de 1 à 20...

```
let x = ref 0 ; ;
for i = 0 to 20 do
x := !x + i ;
done ; ;
!x ; ;
```

On peut aussi descendre en mettant downto à la place de to...

La boucle while

Tant que cette condition est vraie...

```
let x = ref 50_000 ; ;
let i = ref 0 ; ;
while x <> 1 do
incr i ; ;
x :=
if x mod 2 = 0
then !x / 2
else !x * 3 + 1
done ; ;
!i ; ;
```

Quelques structures prédéfinies

Structures modifiables

- 'a ref, 'a array, string

Quelques structures prédéfinies

Structures modifiables

- 'a ref, 'a array, string
- 'a Queue.t, 'a Stack.t

Quelques structures prédéfinies

Structures modifiables

- 'a ref, 'a array, string
- 'a Queue.t, 'a Stack.t
- Buffer.t, ('a,'b) Hashtbl.t, 'a Stream.t, Weak.t

Structures impératives customizées

Déclaration

```
type p_i = { mutable x : int ; mutable y : int } ; ;
```

Utilisation

```
let p = { x = 0 ; y = 0 } ; ;
p.y <- 100 ; ;
p.x,p.y ; ;
```

Tout sur les listes !

Fonctions de base

- length, hd, tl, nth

Tout sur les listes !

Fonctions de base

- length, hd, tl, nth
- rev, append, rev_append, concat

Tout sur les listes !

Fonctions de base

- length, hd, tl, nth
- rev, append, rev_append, concat
- iter, iteri, map, mapi, rev_map, fold_left

Tout sur les listes !

Fonctions de base

- length, hd, tl, nth
- rev, append, rev_append, concat
- iter, iteri, map, mapi, rev_map, fold_left
- for_all, exists, mem, find, filter, partition

Tout sur les listes !

Fonctions de base

- length, hd, tl, nth
- rev, append, rev_append, concat
- iter, iteri, map, mapi, rev_map, fold_left
- for_all, exists, mem, find, filter, partition
- fast_sort, merge

Tout sur les listes !

Fonctions de base

- length, hd, tl, nth
- rev, append, rev_append, concat
- iter, iteri, map, mapi, rev_map, fold_left
- for_all, exists, mem, find, filter, partition
- fast_sort, merge
- et bien d'autres encore...

Appeler un module

Appel direct

```
let l = [1;2;3];;
List.hd l;;
```

Appel de tout un module

```
open List;;
tl l;;
```

Des modules partout !

Les modules qui seront utiles

- Array, String, Random, Printf, Char

Des modules partout !

Les modules qui seront utiles

- Array, String, Random, Printf, Char
- Map, Set, Hashtbl

Des modules partout !

Les modules qui seront utiles

- Array, String, Random, Printf, Char
- Map, Set, Hashtbl
- Buffer, Stack, Queue, Stream

Des modules partout !

Les modules qui seront utiles

- Array, String, Random, Printf, Char
- Map, Set, Hashtbl
- Buffer, Stack, Queue, Stream
- Arg, Sys, Marshal, Filename

Fabriquer rapidement un module

Créer son module

```
module Mod =
struct
type t = int
let f () = 0
let g x = x+1
end ; ;
```

Signer un module

Spécifier la signature

```
module type X =
sig
type t
val f : unit -> t
val g : t -> t
end
```

m

```
odule M : X = Mod ; ;
```