

# Computational effects

11 Exceptions

12 Traits impératifs

13 Continuations

Même si OCaml est un langage fonctionnel, nous avons souvent utilisé dans les exemples passés des caractéristiques qui sortent d'une programmation fonctionnelle pure. C'est le cas de :

- exceptions,
- opérations d'entrée/sortie,
- références,
- continuations explicites.

Nous allons les étudier dans cette partie

11 Exceptions

12 Traits impératifs

13 Continuations

## Typage et domaine de définition

**type inféré  $\neq$  domaine de définition**

- Le typage est une approximation
- Exemples : division entière, tête de liste vide
- provient souvent d'un filtrage non exhaustif

## Que faire ?

- utiliser une valeur spéciale

```
# asin 2.;;  
- : float = nan
```

(\* not-a-number IEEE standard \*)

- effectuer une rupture de calcul jusqu'à un récupérateur

```
# 3/0;;  
Exception: Division_by_zero.
```

## exceptions

```
exception E
```

ou

```
exception E of t;;
```

- une exception est une valeur de type `exn`
- le type `exn` est un type somme monomorphe *extensible*

```
# exception A_MOI;;  
exception A_MOI
```

```
# A_MOI;;  
- : exn = A_MOI
```

```
# exception Depth of int;;  
exception Depth of int
```

```
# Depth 4;;  
- : exn = Depth(4)
```

```
# exception Value of 'a ;;      (* monomorphe *)  
Error: Unbound type parameter 'a
```

## Déclenchement d'une exception

```
# raise;;  
- : exn -> 'a = <fun>
```

- impossible à écrire (primitive)
- l'expression `raise E` n'a pas de contrainte de type

```
# raise A_MOI;;  
Uncaught exception: A_MOI
```

```
# let x = 18;;  
val x : int = 18
```

```
# if (x = 0) then raise A_MOI else x;; (* A_MOI used in int position *)  
- : int = 18
```

## Déclaration et déclenchement d'une exception (2)

### Avec un paramètre

```
# exception Echec of string;;
exception Echec of string

# let declenche_echec s = raise (Echec s);;
val declenche_echec : string -> 'a = <fun>

# declenche_echec "argument invalide";;
Exception: Echec "argument invalide".

# failwith;;
- : string -> 'a = <fun> (*  $\simeq$  fun x -> raise (Failure x) *)
```

# Déclaration et déclenchement d'une exception (3)

## Filtrage de motifs incomplet : déclenchement "involontaire"....

```
# let tete l = match l with t::q -> t;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val tete : 'a list -> 'a = <fun>
```

```
# tete [1;2;3];;
- : int = 1
```

```
# tete [];;
Exception: Match_failure ("", 13, 35).
```

.... ou volontaire afin d'interrompre une execution

```
# exception Found_zero;;
exception Found_zero
```

```
# let rec mult_aux l= match l with
  | h::[] -> h
  | 0::t -> raise Found_zero
  | h::t -> h * mult_aux t;;
Warning: this pattern-matching ...
val mult_aux : int list -> int = <fun>
```

## Syntaxe :

```
try expr with filtrage
```

Le type des motifs du filtrage doit être `exn`.

```
# let mult_list l = match l with
  [] -> 0
| lo -> try mult_aux lo with Found_zero -> 0;;
val mult_list : int list -> int = <fun>

# mult_list [1;2;3;0;5;6];;
- : int = 0
```

## Utilisation des exceptions

- 1 Gestion de situations exceptionnelles où le calcul ne peut pas se poursuivre → rupture du calcul
- 2 style de programmation (par exemple, rupture d'une boucle)

Attention au coût du `try` qui doit sauver le contexte courant

(à placer le plus extérieurement possible surtout s'il y a des boucles)

## Filtrage d'une liste

- filtrage des éléments d'une liste par un prédicat
- **sans recopie inutile**

On veut donc une fonction `filter` qui prend un predicat `p: 'a -> Bool` et une liste `lst` et dont l'implementation

- 1 Retourne immédiatement `lst` si tous les élément de `lst` satisfont `p`
- 2 Si la liste est `h :: t` et `h` satisfait `p`, elle retourne `h :: (filter p t)`, et `(filter p t)` si `h` ne satisfait pas `p`

Utiliser les exceptions pour obtenir le comportement voulu

# Une solution

```
# exception Identity;;
exception Identity

# let filter p l =
  let apply f x = try f x with Identity -> x in
  let rec fil l = match l with
    | [] -> raise Identity
    | h :: t -> if p h then          (* depuis la dernière application de *)
                  h :: fil t       (* apply tout element satisfait p   *)
                else
                  apply fil t in (* on recommence par un apply      *)
  apply fil l;;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

La fonction `fil` fait une copie de la queue de la liste qui satisfait la propriété, mais cette copie est “garbage collectable” à la fin de l’appel de `filter`.

```
# let l1 = .... (* liste avec un important memory footprint *)
# let l2 = filter p l1
```

si tout élément de `l1` satisfait `p` alors `l1` et `l2` dénoteront “physiquement” la même liste en mémoire (c’est équivalent à “`let l1 = l2`”). Plus en général `l1` et `l2` partageront en mémoire toute portion terminale de `l1` qui satisfait `p`.

11 Exceptions

**12 Traits impératifs**

13 Continuations

## Modèle plus proche de la couche physique

- Un bon condensé de programmation impérative :

$$x := x + 1$$

- exécution d'une instruction (action) qui modifie l'état mémoire
  - passage à une nouvelle instruction dans le nouvel état mémoire
- modèle des langages Fortran, Pascal, C, Ada, . . .

- `types : in_channel` ou `out_channel`.

```
# open_in;;  
- : string -> in_channel = <fun>
```

```
# open_out;;  
- : string -> out_channel = <fun>
```

```
# close_in ;;  
- : in_channel -> unit = <fun>
```

```
# close_out ;;  
- : out_channel -> unit = <fun>
```

- `exception : End_of_file`
- canaux prédéfinis : `stdin`, `stdout` et `stderr`
- type `open_flag` pour les modes d'ouverture

```
input          : in_channel → string → int → int → int
input_line    : in_channel → string
output        : out_channel → string → int → int → unit
output_string : out_channel → string → unit
read_line     : unit → string
read_int      : unit → int
print_string  : string → unit
print_int     : int → unit
print_newline : unit → unit
```

## Example : “c’est plus/c’est moins”

```
# let rec cpcm n =
  let () = print_string "taper un nombre : " in
  let i = read_int ()
  in
  if i = n then
    let () = print_string "BRAVO" in
    let () = print_newline ()
    in print_newline ()
  else
    let () =
      if i < n then
        let () = print_string "C+"
        in print_newline ()
      else
        let () = print_string "C-"
        in print_newline ()
    in cpcm n ;;
val cpcm : int -> unit = <fun>
```

```
# cpcm 64;;
taper un nombre : 88
C-
taper un nombre : 44
C+
taper un nombre : 64
BRAVO
```

- valeurs structurées dont une partie peut être physiquement (en mémoire) modifiée ;
- vecteurs, enregistrements à champs modifiables, chaînes de caractères, références

## Attention

Nécessite de contrôler l'ordre du calcul (mais les I/O aussi)

Ça tombe bien : OCaml est *strict*

Une fonction est *stricte* si lorsqu'elle est appliquée à un argument qui ne termine pas, elle ne termine pas. Un langage est *strict* s'il ne peut définir que des fonctions strictes. Un langage avec "eager evaluation" (les variables ne sont liés qu'à des valeurs) est toujours strict.

- regroupent un nombre connu d'éléments de même type
- création : `Array.create : int → 'a → 'a array`,
- longueur : `Array.length : 'a array → int`
- accès : `e1.(e2)`
- modification : `e1.(e2) <- e3`

```
# let v = Array.create 4 3.14;;  
val v : float array = [|3.14; 3.14; 3.14; 3.14|]
```

```
# v.(1);;  
- : float = 3.14
```

```
# v.(8);;  
Exception: Invalid_argument "Array.get".
```

```
# v.(0) <- 100.;;  
- : unit = ()
```

```
# v;;  
- : float array = [|100.; 3.14; 3.14; 3.14|]
```

## Fonctions sur les vecteurs

- création matrice :
  - `Array.make_matrix : int → int → 'a → 'a array array`
- itérateurs :
  - `iter : ('a → unit) → 'a array → unit`
  - `map : ('a → 'b) → 'a array → 'b array`
  - `iteri : (int → 'a → unit) → 'a array → unit`
  - `mapi, fold_left, fold_right, . . .`

# Enregistrements à champs mutables

Dans un enregistrement OCaml il est possible de spécifier des champs qui sont modifiables.

- indication à la déclaration de type d'un champs est "mutable"
- accès au champ identique `e.f`
- modification similaire aux vecteurs `e1.f <- e2`

```
# type point = {mutable x : float; mutable y : float};;  
type point = { mutable x: float; mutable y: float }
```

```
# let p = {x=1.; y=1.};;  
val p : point = {x=1; y=1}
```

```
# p.x <- p.x +. 1.0;;  
- : unit = ()
```

```
# p;;  
- : point = {x=2; y=1}
```

- les chaînes sont des valeurs modifiables (fonction input)
- accès : `e1.[e2]`
- modification : `e1.[e2 ]<-e3`

```
# let s = "bonjour";  
val s : string = "bonjour"
```

```
# s.[3];;  
- : char = 'j'
```

```
# s.[3]<- 't';;  
- : unit = ()
```

```
# s;;  
- : string = "bontour"
```

# Références

On préfère l'utilisation des records mutables, par lesquels ils sont désormais encodés

- `type : 'a ref`  $(\equiv \{\text{mutable contents: 'a}\})$
- `read : !e`  $(\equiv e.\text{contents})$
- `write : e1:=e2`  $(\equiv e1.\text{contents}<-e2)$

```
# let incr x = x := !x + 1;; (* fonction prédefinie *)  
val incr : int ref -> unit = <fun>
```

```
# let z = ref 3;;  
val z : int ref = {contents=3} (* noter le mutable record *)
```

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

```
# z;;  
- : int ref = {contents=4}
```

```
# (ref 3) := 2;;  
- : unit = ()
```

- composition séquentielle : `e1; e2`

il ne s'agit que de sucre syntaxique pour : `let _ = e1 in e2`  
le type de la séquence est le type de `e2`

- `e1` doit être de type `unit`
- Si `e1` n'est pas de type `unit` cela cause un `Warning` :

```
# 1;();;
```

```
Warning S: this expression should have type unit.
```

```
- : unit = ()
```

```
# ignore;;
```

```
- : 'a -> unit = <fun>
```

```
# ignore 1;();;
```

```
- : unit = ()
```

- conditionnelle : `if c then e` (ou `e` est de type `unit`)
- itératives :
  - `while c do e done`
  - `for x=e1 [down]to e2 do e3 done`

La conditionnelle et les boucles sont des expressions de type `unit`

## Exemple : somme de 2 vecteurs

```
#let somme a b =  
  let al = Array.length a and bl = Array.length b in  
  if al <> bl then failwith "somme"  
  else if al = 0 then a  
    else  
      let c = Array.create al a.(0) in  
        for i=0 to al-1 do  
          c.(i) <- a.(i) + b.(i)  
        done;  
        c;;  
val somme : int array -> int array -> int array = <fun>  
  
# somme [|1; 2; 3|] [| 9; 10; 11|];;  
- : int array = [|10; 12; 14|]
```

## Utiliser le bon style selon les structures de données et leurs manipulations (par copie ou en place)

- impératif sur les matrices (en place)
- fonctionnel sur les arbres (par copie)

## Mélanger les deux styles

- valeurs fonctionnelles modifiables
- implantation de l'évaluation retardée

# Example 1 : Map

## En style fonctionnel :

```
# let rec fmap f l = match l with
  [] -> []
  | h::t -> let r = f h in r::(fmap f t);;
val fmap : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

## En style impératif :

```
# let imap f l =
  let nl = ref l
  and nr = ref [] in
  while (!nl <> []) do
    nr := ( f (List.hd !nl) ) :: (!nr);
    nl := List.tl !nl
  done;
  List.rev !nr;;
val imap : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Functional wins !

## Example 2 : Transposée de matrice

### En style impératif :

```
# let itrans m =
  let l = Array.length m in
  for i=0 to l-1 do
    for j=i to l-1 do
      let v = m.(i).(j) in
      m.(i).(j) <- m.(j).(i);
      m.(j).(i) <- v
    done
  done;;
val itrans : 'a array array -> unit = <fun>
```

### En style fonctionnel :

```
# let rec ftransl l = match l with
  []::_ -> []
  | _ -> (List.map List.hd l) :: ftransl (List.map List.tl l);;
val ftransl : 'a list list -> 'a list list = <fun>
```

Imperative wins !

## Example 3 : Simulation d'évaluation paresseuse

### Le calcul est gélé dans un thunk

```
# type 'a thunk = Exp of (unit -> 'a) | Val of 'a;;  
type 'a thunk = Exp of (unit -> 'a) | Val of 'a
```

```
# type 'a delayed = {mutable thunk : 'a thunk};;  
type 'a delayed = { mutable thunk : 'a thunk; }
```

```
# let delay f = { thunk = Exp f };;  
val delay : (unit -> 'a) -> 'a delayed = <fun>
```

```
# let force e = match e.thunk with  
    | Val v -> v  
    | Exp f -> let v = f() in (e.thunk <- Val v ; v);;  
val force : 'a delayed -> 'a = <fun>
```

- Pour retarder l'évaluation de  $e$  il faut écrire `delay(fun() -> e)` (qui a le type `t delayed` si  $e$  est de type `t`)
- Pour forcer l'évaluation de  $e$  il faut écrire `force e` (où  $e$  est de type `t delayed`)
- Toute expression gélée ne sera évaluée qu'une seule fois.

## Example d'évaluation paresseuse

```
# let test1 x = (print_string "pong" ;
                print_newline();
                x+x) in
  let arg = (print_string "ping";
            print_newline();
            6*8) in
  test1 arg;;
ping
pong
- : int = 96
```

```
# let test2 x = (print_string "pong" ;
                print_newline();
                force x + force x) in
  let arg = delay ( fun () -> print_string "ping";
                  print_newline();
                  6*8) in
  test2 arg;;
pong
ping
- : int = 96
```

Winning mix

En Ocaml `force` et `delay` sont respectivement `Lazy.force` et `lazy` :

```
# let test2 x = ( print_string "pong" ;
                  print_newline();
                  Lazy.force x + Lazy.force x) in
  let arg = lazy( print_string "ping";
                  print_newline();
                  6*8) in

  test2 arg;;
pong
ping
- : int = 96
```

## Problèmes

- Beaucoup moins efficace qu'une implémentation native tel que Haskell
- Il n'a pas tous les avantages d'un langage non-strict (e.g. déforestation)
- Même s'il permet la définition de structures paresseuses, il n'a pas la même flexibilité qu'une implantation native.  
(e.g. `test2: int Lazy.t -> int ...` on ne peut pas lui passer un `int`).

11 Exceptions

12 Traits impératifs

**13 Continuations**

La fonction `call/cc` est un opérateur de contrôle qui capture la continuation courante et l'applique à son argument.

Obscur ?

Nous allons l'expliquer en détail lors des transformations de programmes.

Pour l'instant il suffit de savoir qu'elle est implantée en OCaml mais comme dit son auteur (Xavier Leroy) :

*This library implements the `call/cc` (call-with-current-continuation) control operator for Objective Caml. This is a very naive implementation : it works only in bytecode, and performance is terrible (`call/cc` copies the whole stack). It is intended for educational and experimental purposes. Use in production code is not advised.*