# Cours numéro 4 : polymorphisme – filtrage – nuplet – enregistrements

LI213 – Types et Structures de données

Christophe Gonzales – Pierre-Henri Wuillemin

Licence d'Informatique – Université Paris 6

## Fonctions et induction de type

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

Encore mieux :

```
# let a b c = b 5 + b c;;
val a : ... -> ... -> ... = <fun>
val a : ... -> ... -> int = <fun>
val a : (int -> int) -> ... -> int = <fun>
val a : (int -> int) -> int -> int = <fun>
val a : (int -> int) -> int -> int = <fun>
```

Et maintenant :

```
# let horreur b c = b c;;
```

1. `b` est une fonction à 1 paramètre dont le type est celui du second paramètre de `horreur`.
2. Le type du résultat de `b` est le même que celui du type du résultat de `horreur`.



## Polymorphisme (1/2)

*Définition du polymorphisme*

Le polymorphisme consiste à introduire des schémas de type, c'est-à-dire des types avec des variables de type quantifiées universellement.

hum ...

*Idée générale*

```
# let identity x = x;;
```

- quel doit être le type de x pour que cela fonctionne ?
  si x est un char, `identity` devrait rendre un char
  si x est un int, `identity` devrait rendre un int.
- En fait, `identity` devrait fonctionner pour n'importe quel type
  $\Longrightarrow$ polymorphisme.



## Polymorphisme (2/2)

```
# let identity x = x;;
identity : 'a -> 'a
```

Règle : le `'a` signifie $\forall$ *a*, autrement dit, pour tout type.

Le «a» après la quote est très important ! ! ! !

```
# let horreur x y = x y;;
val horreur : ('a -> 'b) -> 'a -> 'b = <fun>
```

*explication :* x y $\Longrightarrow$ x est une fonction. Type : `in -> out`
Pas de contrainte sur y $\Longrightarrow$ y de type `'a`
x prend en entrée y $\Longrightarrow$ `in = 'a`
pas de contrainte sur la sortie de x $\Longrightarrow$ `out` peut être différent du type de y $\Longrightarrow$ `out = 'b`

## Exemples de polymorphisme

```
# (<=);;
- : 'a -> 'a -> bool = <fun>
```

La fonction <= permet de comparer deux éléments de même type, quel que soit ce type.

```
# "blurp" <= "schmurtz";;
- : bool = true
```

Comparaison lexicographique, caractère par caractère, des codes ASCII, etc ...

---

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

permet de calculer $f \circ g\ (x)$

```
# let f x = x * x and g x = x + 1 in o f g 3;;
- : int = 16
```

`f : int -> int` et `g : int -> int`
⇒ `o f g 3` de type `int`



## préliminaire : filtrage de motifs (pattern matching)

Le filtrage par motif s'applique aux valeurs. Il sert à reconnaître la forme de cette valeur et associe à chaque motif une expression à calculer. sorte de *switch* mais en plus fort !

### switch et match

```
switch (varMois) {
  case 1 :
     return 'j';
     break;
  case 2 :
     return 'f';
     break;
  default :
     return 'a';
     break;
}
```

```
match valMois with
  | 1 -> 'j'

  | 2 -> 'f'

  | _ -> 'a'

;;
```



## filtrage de motifs (2)

### Combinaison de motifs

```
# let est_une_voyelle c = match c with
  'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
  | _ -> false
;;
```

⚠ Le filtrage est une opération de transformation de valeurs.

```
# let f x = match x with 2 -> true | _ -> false;;
val f : int -> bool = <fun>
# let f x = match x with 2 -> true;;
Warning : this pattern-matching is not exhaustive.
Here is an example of a value that is not matched : 0
# f 1;;
Uncaught exception : Match_failure ...
```



## filtrage de motifs (3) : filtrage de paramètres

### est_un_ou_deux (v1)

```
# let est_un_ou_deux x =
    match x with 1 | 2 -> true | _ -> false;;
val est_un_ou_deux : int -> bool = <fun>
```

### est_un_ou_deux (v2) : utilisation de fonctionnelle

```
# let est_un_ou_deux = function x ->
    match x with 1 | 2 -> true | _ -> false;;
val est_un_ou_deux : int -> bool = <fun>
```

### est_un_ou_deux (v3) : filtrage explicite de paramètre

```
# let est_un_ou_deux =
    function 1 | 2 -> true | _ -> false;;
val est_un_ou_deux : int -> bool = <fun>
```

## Les types complexes : agrégation de données

Il est régulièrement nécessaire de travailler dans un programme avec un nombre important de données.

- **représentation de données multidimensionnelles**
  - espace euclidien, par exemple 3D : (1.5,3.2,4.25)
  - espace “bizarre” : réponses à un questionnaire (13,8.5,’o’,’n’,’o’,’o’)
  - ...

  n-uplets
- **caractérisation complexe d’un objet**
  - un individu (nom/prénom/âge/etc.)
  - un descripteur de fichier (nom/taille/droits/etc.)
  - ...

  enregistrements
- **répétitions en nombre d’un même type d’objet**
  listes et tableaux.



## Les n-uplets ou produit cartésien

**type produit** : mécanisme d’agrégation standard de OCaml.

*Math* : Si $x \in \mathcal{X}$ et $y \in \mathcal{Y}$ alors $(x, y) \in \mathcal{X} \times \mathcal{Y}$

Un n-uplet est une valeur qui s’écrit sous la forme :

```
(valeur1, valeur2, ...)
```

### Exemples

```
# (3.0 , 2);;
- : float * int = (3., 2)
#  23,true;;
- : int * bool = (23, true)
# let o3D=(0 , 0 , 0);;
val o3D : int * int * int = (0, 0, 0)
```



 Une virgule forme un n-uplet

### Exemple

```
# let f x y = x + y ;;
val f : int -> int -> int = <fun>
# f 3 5;;
- : int = 8
# f 3 ;;
- : int -> int = <fun>
# f(3,5);;
This expression has type int * int but is here
used with type int
# f 3,4;;
- : (int -> int) * int = (<fun>, 4)
```



## Filtrage sur les n-uplets

### primitives sur les couples (2-uplets)

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# snd;;
- : 'a * 'b -> 'b = <fun>
```

```
# fst (4,true);;
- : int = 4
# snd (4,true);;
- : bool = true
```

### Définition de fst

```
# let f couple = match couple with (x,y) -> x;;
val f : 'a * 'b -> 'a = <fun>
```

Le filtrage peut “dé-structurer” un n-uplet.

```
# let f (x,y) = x;;
val f : 'a * 'b -> 'a = <fun>
```

## Filtrage sur les n-uplets : filtrage universel

### Définitions du OU logique

```
# let ou_logique (a,b) = match (a,b) with
  | (true,true) | (true,false)
  | (false,true) -> true
  | (false,false) -> false ;;
# let ou_logique (a,b) = match (a,b) with
  | ( true, _ ) -> true
  | (false,true) -> true
  | (false,false) -> false ;;
# let ou_logique (a,b) = match (a,b) with
  | ( true, _ ) -> true
  | ( false, _ ) -> b ;;
# let ou_logique (a,b) = match (a,b) with
  | (false,false) -> false
  | _ -> true ;;
# let ou_logique =
  function (false,false) -> false | _ -> true ;;
```



## Filtrage sur les n-uplets : liaisons

### IMPORTANT

Les identifiant qui apparaissent dans des motifs sont *forcément non liés*.
Le filtrage correspond exactement à lier localement ces identifiants à des parties de la valeur filtrée.

```
# let f cpl = let a=2 and b=3 in
              match cpl with (a,b) -> a+b;;
val f : int * int -> int = <fun>
# f (4, 5);;
- : int = 9
# let est_egal x y =
    match x with y -> true | _ -> false;;
  Warning : this match case is unused.
val f : 'a -> 'b -> bool = <fun>
```

Retourne toujours `true`



## Filtrage sur les n-uplets : améliorations

### Filtrage sans liaisons inutiles

```
# let fst couple = match couple with (x,y) -> x;;
val fst : 'a * 'b -> 'a = <fun>
# let third t = match t with (_,_,x)->x;;
val third : 'a * 'b * 'c -> 'c = <fun>
```

### Filtrage avec garde

```
# let est_double_non_nul x y = match (x,y) with
    (0,0) -> false
  | (a,b) when b=2*a -> true
  | _ -> false ;;
```

Le filtrage n'est accepté que si la condition de garde est vérifiée.



## Filtrage sur les n-uplets : liaisons (2)

### IMPORTANT

Les identifiant qui apparaissent dans des motifs ne peuvent être liés qu'une fois

```
# let fst_egal_snd c = match c with
  (a,a) -> true
  | _ -> false;;
This variable is bound several times
in this matching
```

### solution avec garde

```
# let fst_egal_snd c = match c with
  (a,b) when a=b -> true
  | _ -> false;;
val fst_egal_snd : 'a * 'a -> bool = <fun>
```

## Préambule : Déclaration de types

Possibilité d'étendre les possibilités de typages de OCaml.

*Type, Type paramétré*

```
# type nom = définition;;
# type ('a,'b,'c,...) nom = définition avec 'a,...;;
```

```
# type couple_e_c = int * char;;
type couple_e_c = int * char
# let e=(3,'a');;
val e : int * char = (3, 'a')
# let (e : couple_e_c) = (3,'a');;
val e : couple_e_c = (3, 'a')
# type 'autre couple_e = int * 'autre;;
# type couple_e_c = char couple_e;;
```



## Les enregistrements

*enregistrement (cf. `record` de Ada ou `struct` de C)*

N-uplets dont chaque champ est nommé.
Un enregistrement correspond toujours à la déclaration d'un nouveau type.

```
# type nom = { c1 : type1; c2 : type2; ...};;
```

```
# type complex = { re :float; im :float } ;;
type complex = { re : float; im : float }
```



## Valeurs d'enregistrements (2)

*Valeur de type enregistrement*

```
# { c1 = v1; c2 = v2; ...};;
```

```
# let c= {re=2.; im=3.};;
val c : complex = {re=2.; im=3.}
# c = { im=3.; re=2.};;
- : bool = true
# let d = { im=4. } ;;
Some labels are undefined
```



## Accès aux champs

*Accès aux champs : notation pointée*

```
# expr.nom_de_champs;;
```

```
# let c= {re=2.; im=3.};;
# c.re;;
- : float = 2.
# let f x = x.re;;
val f : cplx -> float = <fun>
```

**Bizarre**

```
# type cplx={re :float;im :float};;
# type ploc={re :int;toto :bool};;
# let f x = x.re;;
val f : ploc -> int = <fun>
```

## Exemple : complexe

```
# type cplx={re :float;im :float};;
# let somme a b =
    {re=a.re +. b.re; im=a.im +. b.im};;
val somme : cplx -> cplx -> cplx = <fun>
# let modul a =
    sqrt (a.re*.a.re +. a.im*.a.im);;
val modul : cplx -> float = <fun>
# let make_cplx a b = {re=a;im=b} ;;
val make_cplx : float -> float -> cplx = <fun>
# let inflate x c =
    make_cplx (x*.c.re) (x*.c.im);;
val inflate : float -> cplx -> cplx = <fun>
```



## filtrage sur les enregistrements

Filtrage d'un enregistrement : nommer la valeur de champs.

### Accès aux champs par notation pointée

```
# let add_cpl z1 z2 = {
  re=z1.re +. z2.re;
  im=z1.im +. z2.im};;
# let mult_cpl z1 z2 = {
  re=z1.re *. z2.re -. z1.im *. z2.im;
  im=z1.re *. z2.im +. z1.im *. z2.re};;
```

### Accès aux champs par filtrage

```
# let mult_cpl z1 z2 = math (z1,z2) with
    ({re=x1;im=y1},{re=x2;im=y2}) -> {
  re=x1 *. x2 -. y1 *. y2;
  im=x1 *. y2 +. y1 *. x2};;
```



## Retour sur le filtrage de motifs

### *Rappel de définition*

```
match expr with
 |  motif1 -> exp1
 |  motif2 -> exp2
        ...
```

- Un *motif* ou *pattern* est une expression qui représente une forme possible de la valeur *expr*.
- Les valeurs `exp1, exp2, ...` doivent être de **même type**. Le `match` est considéré comme une expression ayant ce même type.
- La valeur de cette expression sera celle associée au premier *motif* qui filtrera correctement *expr*.
- Un *motif* ou *pattern* peut contenir des **identifiants non liés** qui seront liés localement aux parties de *expr* identifiés lors de l'évaluation de l'expression associée.



## Filtrage - sans liaison locale

Filtrage simple sans liaisons locales dans les motifs (les motifs sont des constantes).

```
#  let f n =
    match n with
        0 -> "zéro"
      |  1 -> "un"
      |  2 | 3 | 4 | 5 -> "de 2 à 5"
      | _ -> "plus grand que 5"
  ;;
val f : int -> string = <fun>
# f 0;;
- : string = "zéro"
# f 150;;
- : string = "plus grand que 5"
```

## Filtrage - Liaisons locales : Calcul sans intérêt de x/2+y*2

```
# let f n = match n with
    (0,x) -> x *. 2.
  | (x,0.) -> float_of_int ( x / 2)
  | (x,y) -> float_of_int ( x / 2) +. y *. 2.
;;
val f : int * float -> float = <fun>
# f (2,3.);;
- : float = 7.
```

1. Évaluation de `f (2,3.)` dans $E_0$.
2. Évaluation du `match` dans E_local= (n,(2,3.)) ▷$E_0$
3. Filtrage incorrect pour `(0,x)` et `(x,0.)`
4. Filtrage OK pour `(x,y)`
5. Évaluation de `float_of_int ( x / 2) +. y *. 2.` dans (x,2),(y,3.)▷E_local



## Filtrage - Fibonacci et filtrage de paramètre

La fonction de calcul de terme de la suite de Fibonacci s'écrit naturellement en récursif avec un match :

```
# let rec fibo n = match n with
    0 -> 1
  | 1 -> 1
  | n -> fibo (n-1) + fibo(n-2)
;;
val fibo : int -> int = <fun>
```

⚠ `fibo(n-1) +fibo(n-2)` est évalué dans un environnement contenant **2 liaisons pour n :**

1. la liaison locale à la fonction : l'argument de la fonction
2. la liaison locale au match : le filtrage de l'expression «n»



## Filtrage - Fibonacci et filtrage de paramètre (2)

On peut réécrire `fibo` sans match mais avec filtrage explicite de paramètre :

```
# let rec fibo = function
    0 -> 1
  | 1 -> 1
  | n -> fibo (n-1) + fibo(n-2)
;;
val fibo : int -> int = <fun>
```



## Filtrage - Filtrage et décomposition

- Un filtrage permet de décomposer des agrégations de données (n-uplet, enregistrement, etc...)

```
# let troisieme x =
    match x with
      (u,v,w) -> w;;
val troisieme : 'a * 'b * 'c -> 'c = <fun>
```

⚠ Le motif a contraint le type de x (triplet)

- Les liaisons inutiles peuvent être remplacées par un _

```
# let troisieme x =
    match x with
      (_,_,u) -> u;;
val troisieme : 'a * 'b * 'c -> 'c = <fun>
```

- En utilisant le filtrage de paramètre

```
# let troisieme = function (_,_,u) -> u;;
val troisieme : 'a * 'b * 'c -> 'c = <fun>
```

## filtrage sur les enregistrements (2)

Le filtrage n'a pas à être complet.

### filtrage incomplet

```
# let partie_imaginaire z =
    match z with {im = i} -> i;;
```

### Modification partielle d'enregistrement

```
# z;;
- : complex = {re=2.; im=3. }
# let z1={z with im=5.};;
val z1 : complex = {re=2.; im=5.}
```



## Enregistrement mutable : modification des champs

Les champs des enregistrements peuvent être modifiés par des affectations pourvu qu'ils aient été déclarés `mutable` lors de la définition de l'enregistrement.

### mutable

```
# type point_mutable =
   {mutable x : float;mutable y : float};;
type point_mutable =
   { mutable x : float; mutable y : float }
# let monpoint = {x=0.;y=0. };;
val monpoint : point_mutable = {x=0.; y=0.}
# monpoint.x <- 3.;;
```


affectation

```
- : unit = ()
# monpoint;;
val monpoint : point_mutable = {x=3.; y=0.}
```



## Enregistrement mutable : valeur ou variable ?

- Par un `let`, un identifiant est bien lié à une **valeur** de type enregistrement.
- Cette valeur ne peut pas changer.
- Mais une valeur de type enregistrement n'est pas définie par le contenu des champs.
- Une valeur de type enregistrement n'est pas modifiée par la modification d'un de ces champs.

### exemple

```
# type individu = {...;mutable age :int;...};;
# let pierre={...; age=23; ...};;
# let fetesSonAnniversaire i = i.age<-i.age+1;;
# feteSonAnniversaire pierre;;
```

Pierre est toujours la même personne, mais son âge a changé ...

On utilisera le symbole `<-` également pour les tableaux. Ce sera l'occasion de reparler de tout ça...



## N-uplet ou enregistrement ?

### *Avantages des N-uplets*

- Syntaxe plus légère.
- Construction de N-uplets "anonyme" donc plus rapide :

```
match (x,y,z) with ...
```

versus

```
match {i1=x; i2=y; i3=z} ...
```

### *Avantages des enregistrements*

- une information (sémantique) descriptive et discriminante grâce aux noms des champs.
- un accès identique, par son nom, de n'importe quel champ de l'enregistrement : l'ordre des champs n'a plus d'importance, seul leur nom compte.
- enregistrements mutables.