Introduction à la programmation fonctionnelle

Cours 3

kn@lmf.cnrs.fr
https://usr.lmf.cnrs.fr/~kn

université
PARIS-SACLAY

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions ✓

2 IPF (2) : fonctions récursives, inférence de types ✓

3 IPF (3) : Types structurés, filtrage, polymorphisme, ordre supérieur

- 3.1 Types structurés
- 3.2 Filtrage par motifs
- 3.3 Polymorphisme
- 3.4 Ordre supérieur

Type structuré

Un **type structuré** est un type permettant d'associée plusieurs données de façon à les traîter comme un tout.

Un exemple de type structuré vu en L1 est le tableau :

- collection ordonnée d'éléments, de taille fixe
- les éléments sont accessibles par décalage par rapport au premier élément (indice)
- les « cases » du tableau sont modifiables

Nous allons voir plusieurs types structurés proposés en standard par OCaml et adaptés à la programmation fonctionnelle.

Type produit

Le type structuré le plus simple est celui des produits, aussi appelés n-uplets.
En OCaml, une expression n-uplet se note `(`e_1`, …, `e_n`)` :

```
let div_mod a b =
   (a / b, a mod b)
```

Le type des n-uplets se note t_1 `* … *` t_2. Le nom de produit vient du « produit Cartésien » $A \times B$ entre deux ensembles.
La fonction `div_mod` ci-dessus a le type

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

Elle prend en argument deux entiers et renvoie une paire d'entiers.

Type produit (2)

Une façon commode de travailler avec les n-uplet est d'utiliser un `let` multiple.

```
let x, y = div_mod 10 3
let () = Printf.printf "%d/%d = %d et il reste %d\n" 10 3 x y
```

Une autre façon de faire est d'utiliser les fonctions prédéfinies : `fst` (pour *first*) et `snd` (pour *second*) :

```
let res = div_mod 10 3
let () = Printf.printf "%d/%d = %d et il reste %d\n"
             10 3 (fst res) (snd res)
```

Type produit (3)

Les n-uplets ne sont pas limités aux paires :

```
let hms s =
    let h = s / 3600 in
    let s = s / 3600 in
    let m = s / 60 in
    let s = s mod 60 in
      (h, m, s)

(* val hms : int -> int * int * int *)
```

Attention, dans ce cas on ne peut plus utiliser les fonctions `fst` et `snd` qui sont réservées aux paires. La notation `let` est à privilégier :

```
let h, m, s = hms 4932
let () = Printf.printf
 "%d secondes font %d heure(s), %d minute(s) et %d seconde(s)\n"
   4932 h m s
(*Affiche:
4932 secondes font 1 heure(s) 22 minute(s) et 12 seconde(s)
*)
```

Produit nommé

Une variante du type produit est le **produit nommé** aussi appelé **enregistrement** (ou *struct* ou *record*).
En OCaml, un produit nommé doit être déclaré au moyen de l'instruction `type` :

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

let origin = { x = 0.0; y = 0.0 }

let dist p1 p2 =
  let dx = p1.x -. p2.x in
  let dy = p1.y -. p2.y in
  sqrt (dx *. dx + dy *. dy)

(* dist : point -> point -> float *)

let p1 = { x = -1.0; y = 5.5 }
let p2 = { x = 10.0; y = 3.4 }
```

Produit nommé (2)

La syntaxe pour définir un produit nommé est :

```
type nom_type = {
  lab₁ : type₁ ;
    …
  labₙ : typeₙ ; (* le dernier ; est optionnel *)
}
```

Les $\texttt{lab}_i$ sont des étiquettes ou des champs. Ils permettent d'accéder aux composantes du produit nommé. Les expressions s'écrivent :

```
{
  lab₁ = e₁ ;
    …
  labₙ = eₙ ; (* le dernier ; est optionnel *)
}
```

On peut ensuite accéder à un champs par : $\texttt{e.lab}_i$.

Produit nommé (3)

Les produits nommés sont un peu moins souple que les n-uplets :

- On doit d'abord définir le type avant de l'utiliser
- Deux types différents ne peuvent pas partager le même nom de champ.

```
type personne = { nom : string; prenom : string }
type pays = { nom : string; lat : float; long : float }

let get_nom p = p.nom
(* de type:
 pays -> string
 l'étiquette nom du types pays « masque » celle du type personne.
*)
```

Pour les TPs, on utilisera toujours des noms d'étiquettes uniques.

Type sommes

Il est courant en informatique d'avoir un type composé de plusieurs « cas » différents. Pour gérer cela, OCaml propose des types « sommes » (aussi appelés types algébriques ou variants). On va prendre l'exemple d'un jeu de cartes que l'on souhaite modéliser :

```
type couleur = Coeur | Pique | Trefle | Carreau
```

Le code ci-dessus permet de définir quatres constantes, du type couleur. C'est une approche plus robuste que d'utiliser des entiers ou des chaînes de caractères. On veut aussi pouvoir définir la valeur d'une carte :

```
type valeur = Roi | Dame | Valet | Valeur of int
```

Ici, on indique que la valeur d'une carte peut être soit l'une des trois constantes `Roi`, `Dame`, `Valet` soit une un entier « décoré » par l'étiquette `Valeur`. On peut enfin définir une carte, par exemple en utilisant un produit nommé :

```
type carte = { valeur : valeur ; couleur : couleur }
let roi_pique = { valeur = Roi; couleur = Pique }
let sept_coeur = { valeur =  Valeur (7); couleur = Coeur  }
```

Type sommes (2)

On peut manipuler les types sommes comme n'importe quelle valeur OCaml :

```
let est_tete c =
  c.valeur = Roi || c.valeur = Dame || c.valeur = Valet

(* est_tete : carte -> bool *)
```

Cette utilisation est cependant inélégante. De plus, on ne peut pas utiliser les « entiers étiquetés » directement :

```
let as_trefle = { valeur = Valeur 1; couleur = Trefle}
let () = Printf.printf "valeur : %d\n" as_trefle.valeur
(*
Error: This expression has type valeur but an expression was
       expected of type int
*)
```

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions ✓

2 IPF (2) : fonctions récursives, inférence de types ✓

3 IPF (3) : Types structurés, filtrage, polymorphisme, ordre supérieur

- 3.1 Types structurés ✓
- 3.2 Filtrage par motifs
- 3.3 Polymorphisme
- 3.4 Ordre supérieur

Filtrage ?

Les types produits (n-uplets ou produits nommés) possèdent une notation simple pour accéder à leur composantes (`let x, y = …`, `e.x`, ...).
Comment manipuler des valeurs d'un type somme ?

```
let string_of_valeur v =
      match v with
      Roi -> "Roi"
      | Dame -> "Dame"
      | Valet -> "Valet"
      | Valeur (1) -> "As"
      | Valeur (n) -> string_of_int n

(* string_of_valeur : valeur -> string *)
```

Dans un type sommes, il faut tester tous les cas possibles (étant donné une `valeur` de carte, on ne sait pas a priori dans quel cas on est). Cette construction est similaire au `switch` de C/C++/Java mais est plus sophistiquée.

Construction `match with`

Un type somme peut être inspecté en OCaml au moyen de la construction `match … with`. Cette dernière a la syntaxe suivante :

```
match e with
  p1 -> e1
| p2 -> e2
  …
| pn -> en
```

`e` est l'expression à analyser. Chaque p_i `->` e_i est appelé une branche. Les p_i sont appelés des motifs (*pattern* en anglais). Les e_i sont des expressions.

En première approximation, un motif est soit une constante d'un type somme, soit une valeur étiquetée. Dans ce cas on peut capturer des sous-valeurs au moyen de variables (comme le `n` dans le transparent précédent).

Construction `match with` (2)


```
match e with
  p1 -> e1
| p2 -> e2
  …
| pn -> en
```

Dans l'expression ci-dessus, `e` est évalué pour donner une valeur `v`. Puis, `v` est comparée tour à tours aux motifs :

- si `v` est compatible avec p_1, alors e_1 est évalué
- sinon si `v` est compatible avec p_2, alors e_2 est évalué
- sinon si `v` est compatible avec p_3, alors e_3 est évalué
- …
- sinon `v` est compatible avec p_n e_n est évalué

Exhaustivité du filtrage

Le compilateur OCaml détecte si un filtrage est incomplet ou au contraire redondant. Dans les deux cas un « warning » est émis :

```
# let dix = Valeur (10);;
val dix : valeur = Valeur 10
# match dix with
  Roi -> "Roi"
  | Valeur (n) -> string_of_int n;;
Warning 8: this pattern-matching is not exhaustive.
  Here is an example of a case that is not matched:
  (Dame|Valet)
# match dix with
  Roi -> "Roi"
  | Dame -> "Dame"
  | Valet -> "Valet"
  | Valeur (n) -> string_of_int n
  | Valeur (1) -> "As";;
Warning 11: this match case is unused.
```

On fera en sorte de toujours avoir du code sans *warning*.

Exemples complets

On illustre l'utilisation du filtrage avec d'autres fonctions

```
let string_of_couleur c =
       match c with
       Pique -> "Pique"
       | Coeur -> "Coeur"
       | Trefle -> "Trefle"
       | Carreau -> "Carreau"

(* string_of_couleur : couleur -> string *)

let string_of_carte c =
  (string_of_valeur c.valeur) ^ " de " ^ (string_of_couleur c.couleur)

let as_pique = { valeur = Valeur (1); couleur = Pique}

let s = string_of_carte as_pique
(* s : string = "As de Pique"  *)
```

Exemples complets (2)


```
(* renvoie -1 si c1 < c2, 0 si c1 = c2 et 1 si c1 > c2 *)
let compare_cartes c1 c2 =
  match c1.valeur, c2.valeur with
  | Roi, Roi | Dame, Dame | Valet, Valet -> 0
  | Roi, _ -> 1
  | _  , Roi -> -1
  | Dame, _ -> 1
  | _, Dame -> -1
  | Valet, _ -> 1
  | _, Valet -> -1
  | Valeur v1, Valeur v2 ->
      if v1 < v2 then -1
      else if v1 = v2 then 0
      else 1

(* compare_cartes : carte -> carte -> int  *)
(* Pas de warning, on sait qu'on a traité tous les cas *)
```

Remarque : _ est une variable spéciale qui signifie «n'importe quelle valeur».
On peut factoriser les motifs qui ont la même expression: p_1 `|` p_2 `|` p_3 `->` e

Activité: modélisation de formes

On veut modéliser des formes (triangle, rectangle, carré, cercle, ellipse) de façon à pouvoir calculer leur aire.

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions ✓
2 IPF (2) : fonctions récursives, inférence de types ✓
3 IPF (3) : Types structurés, filtrage, polymorphisme, ordre supérieur
- 3.1 Types structurés ✓
- 3.2 Filtrage par motifs ✓
- 3.3 Polymorphisme
- 3.4 Ordre supérieur

Fonctions `fst` et `snd`

On revient un moment sur les fonctions `fst` et `snd` qui permettent d'extraire la première et seconde composante d'une paire.
Elles sont définies dans la bibliothèque standard d'OCaml avec un simple filtrage :

```
let fst p =
    match p with
      (x, _) -> x

let snd p =
    match p with
      (_, y) -> y

let p1 = (1, 2)
let x1 = fst p1  (* x1 : int = 1 *)
let p2 = (true, "Hello")
let y2 = snd p2 (* y2 : string = "Hello" *)
```

Quel est le type de `fst` et `snd` ?

Fonctions `fst` et `snd` (2)

Essayons de simuler le compilateur OCaml et écrivons les équations que l'on peut obtenir en typant `fst` :

```
let fst p =
    match p with
      (x, _) -> x
```

- `p` : X_p, `fst` : X_p->X_{fst}
- X_p = X_a*X_b car `p` est comparé à un motif de paire.
- X_{fst} = X_a car la fonction renvoie `x` la première composante de la paire.
- C'est tout ! Il n'y a pas d'autre contrainte (`x` n'est pas comparé à un entier, on ne fait pas `x-1`, ...)

Le type final de `fst` est donc : X_a*X_b->X_a
C'est exactement ce qu'OCaml nous affiche :

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

Polymorphisme

Une fonction est dite **polymorphe** si le type de ses arguments ou de ses résultats n'est pas contraint. OCaml signale de tels types par : `'a`, `'b`, `'c`, ... (on lit α, β, γ, ...)

```
# fst;;
val fst : 'a * 'b -> 'a = <fun>
# snd;;
val snd : 'a * 'b -> 'b = <fun>
# let id x = x;;
val id : 'a -> 'a = <fun>
# let dup x = (x, x);;
val dup : 'a -> 'a * 'a = <fun>
```

Polymorphisme (2)

De telles fonctions peuvent être appliquées à n'importe quelles valeurs si les types sont compatibles :

```
# fst (1, 2);;
- : int = 1
# snd (true, "Hello");;
- : string = "Hello"
# id 42.0
- : float = 42.0
# dup false;;
- : bool * bool = (false, false)
```

On peut appliquer `fst` à n'importe quel type paire.

Note : c'est équivalent aux *variables génériques* du langage Java `HashMap<K, V>`

Plan

1 IPF (1) : expressions de bases, if/then/else, fonctions ✓

2 IPF (2) : fonctions récursives, inférence de types ✓

3 IPF (3) : Types structurés, filtrage, polymorphisme, ordre supérieur

- 3.1 Types structurés ✓
- 3.2 Filtrage par motifs ✓
- 3.3 Polymorphisme ✓
- 3.4 Ordre supérieur

Factorisation de code ?

Une des affirmations de ce cours est que la programmation fonctionnelle est un outil permettant d'écrire du code plus compact. Considérons les trois fonctions :

```
let rec fact n =
   if n = 0 then
    1
   else
    n * fact (n-1)

let res = fact 10
(* 3628800 *)
```

```
let rec sum n =
   if n = 0 then
    0
   else
    n + sum (n-1)

let s = sum 10;;
(* 55 *)
```

```
let rec to_str n =
   if n = 0 then
    "0"
   else
    (string_of_int n)
    ^ " " ^
    to_str (n-1)

let txt = to_str 10
(* "10 9 8 7 6 5 4 3 2 1 0" *)
```

On souhaite ne pas dupliquer le code contrôle (`if`, appel récursif, ...)

Quels points communs ?

On veut isoler les points communs de ces trois fonctions :

- récursives sur un argument entier (`n`)
- cas de base sur `n = 0`
- Une valeur spéciale est renvoyée dans le cas de base (`0`, `1`, `"0"`)
- Dans le cas récursif on effectue une opération entre la valeur courante `n` et le résultat de l'appel récursif.

On veut pouvoir écrire une fonction `recur_gen` qui a ce comportement. La fonction doit être paramétrée par :

- `n` (évident)
- la valeur de base
- l'opération de combinaison entre la valeur courante et le cas de base

Comment représenter cette opération ? Avec une fonction !

Fonction générique


```
let rec recur_gen base op n =
   if n = 0 then
    base
   else
    op n (recur_gen base op (n-1))
```

Quel est le type de `recur_gen` ?

```
'a -> (int -> 'a -> 'a) -> int -> 'a
```

- Le premier argument est d'un type quelconque `'a`, celui de la valeur de base
- Le deuxième argument est une fonction, celui de l'opération. Cette fonction prend deux arguments, un `int` (la valeur courante de `n`) et le résultat de l'appel récursif. Celui-ci doit être du même type que le case de base, donc `'a`. Le résultat de l'opération est renvoyé dans le cas `else`, il doit donc être du même type que le cas de base aussi.
- Le troisième argument est l'entier `n`
- Le résultat de la fonction doit être du même type que le cas de base `'a`

Comment spécialiser ?


```
let add x y = x + y
(* add : int -> int -> int *)

let mul x y = x * y
(* mul : int -> int -> int *)

let add_str x y = (string_of_int x) ^ " " ^ y
(* add_str : int -> string -> string *)

let fact n = recur_gen 1 mul n  (* int -> int *)

let sum n = recur_gen 0 add n   (* int -> int *)

let to_str n = recur_gen "0" add_str n (* int -> string *)
```

Spécialisation de fact (exemple)


```
recur_gen 1 mul n
```

↓

```
let rec recur_gen base op n =
  if n = 0 then
    base 1
  else
  op mul n (recur_gen base op (n-1))
```

↓

```
let rec recur_gen n =
  if n = 0 then
    1
  else
    n * (recur_gen (n-1))
```

Bonus

En OCaml les opérateurs (`+`, `+.`, `^`, ...) peuvent aussi être utilisées comme des fonctions. On écrit alors : `( + )`, `( +. )`, `( ^ )`, ...

```
# ( + );;
val ( + ) : int -> int -> int = <fun>
# ( +. )
val ( +. ) : float -> float -> float = <fun>
# ( ^ )
val ( ^ ) : string -> string -> string = <fun>
#  ( && )
val ( && ) : bool -> bool -> bool = <fun>
```

On peut définir `fact` et `sum` de façon encore plus compacte.

```
let fact n = recur_gen 1 ( * ) n  (* int -> int *)

let sum n = recur_gen 0  ( + ) n  (* int -> int *)
```

Ordre supérieur

- On peut factoriser du code en écrivant des fonctions génériques qui prennent en argument d'autres fonctions
- Une fonction prenant en argument une autre fonction est dite « d'ordre supérieur »

C'est un outil très puissant que l'on va réutiliser par la suite :

- Trier une collection en passant en paramètre la fonction de comparaison
- Afficher une collection en passant en argument la fonction permettant d'afficher les éléments
- « Aggréger » une collection (calculer la somme, la moyenne, la taille, ...) d'une collection
- ...

Activités

Note: un prédicat sur les entiers est une fonction de type `int -> bool`

- Écrire une fonction `find` qui prend en argument un prédicat, et 2 bornes entières `i ≤ j` et renvoie le premier entier compris entre les bornes qui vérifie le prédicat.
- Écrire une fonction `make_printer` qui prend en argument un prédicat et renvoie ce même prédicat mais qui affiche en plus son entrée et sa sortie sur la sortie standard

Donner le type de `find` et `make_printer`