Programmation Fonctionnelle
Cours 4
D'autres types structurés

Delia Kesner

Table de matières

Concepts de base

- Le type produit cartésien type un *n*-uplet
- Un *n*-uplet `(e1,..,en)` a un type `t1*...*tn`,
  si chaque élément `ei` est de type `ti`.
- Ne pas confondre les *n*-uplets avec les listes !!
  - un *n*-uplet est de longueur fixe *n*
  - une liste est de longueur variable
  - un *n*-uplet peut contenir des éléments de type diffèrent
  - une liste contient des éléments du même type
- Utilité:
  - écrire des fonctions qui prennent un *n*-uplet en argument (ne pas à confondre avec les fonctions à plusieurs arguments !)
  - écrire des fonctions qui renvoient plusieurs résultats groupés sous forme d'un *n*-uplet
  - Construire des types plus complexes:
    - les listes d'association
    - les types algébriques

Tuples I

```
(2, 3, 4);;
# - : int * int * int = (2, 3, 4)
(1, 3.0, "hello");;
# - : int * float * string = (1, 3., "hello")

let pair = (1,2);;
# val pair : int * int = (1, 2)
let liste = [1;2] ;;
# val liste : int list = [1; 2]

pair = liste;;
pair = liste;;
       ^^^^^
Error: This expression has type int list
       but an expression was expected of type int * int
```

Construire et déconstruire des n-uplets

- Pour construire : il faut combiner les éléments par la virgule (souvent écrit entre parenthèses, mais ce n'est pas strictement nécessaire) : `(1,2.0,"hello")`
- Pour déconstruire : Il y a des fonctions `fst` et `snd` pour les paires, sinon le filtrage par motif ...
- ... ou avec un let : plus simple que le filtrage par motifs en général car il y a un seul cas

2-uplet I

```
(*construction*)
let nom_age = ("Jean",25);;
# val nom_age : string * int = ("Jean", 25)
let prenom_nom = ("Jean","Dupont");;
# val prenom_nom : string * string = ("Jean", "Dupont")

(*deconstruction: premiere projection *)
fst nom_age;;
# - : string = "Jean"

(*deconstruction: seconde projection *)
snd nom_age;;
# - : int = 25
```

2-uplet II

```
(* fonctions de projection alternatives *)
let fst' z = match z with (x,y) -> x;;
# val fst' : 'a * 'b -> 'a = <fun>
let fst' z = let (x,y) = z in x;;
# val fst' : 'a * 'b -> 'a = <fun>
let snd' z = match z with (x,y) -> y;;
# val snd' : 'a * 'b -> 'b = <fun>
let snd' z = let (x,y) = z in y;;
# val snd' : 'a * 'b -> 'b = <fun>
```

3-uplet I

```
(*construction*)
let nom_prenom_age = ("Marie","Dupont",25);;
# val nom_prenom_age : string * string * int =
  ("Marie", "Dupont", 25)

(* fonctions de projection *)
let proj1 z = let (x,y,w) = z in x;;
# val proj1 : 'a * 'b * 'c -> 'a = <fun>
let proj2 z = let (x,y,w) = z in y;;
# val proj2 : 'a * 'b * 'c -> 'b = <fun>
let proj3 z = let (x,y,w) = z in w;;
# val proj3 : 'a * 'b * 'c -> 'c = <fun>
```

Fonction avec un seul argument ou avec deux arguments? I

```
(* Fonction avec un argument *)
let f(x,y) = x+y;;
# val f : int * int -> int = <fun>
f(2,3);;
# - : int = 5
f 2 3;;
f 2 3;;
^
Error: This function has type int * int -> int
       It is applied to too many arguments;
       maybe you forgot a ';'.
```

Fonction avec un seul argument ou avec deux arguments? II

```
(* Fonction avec deux arguments *)
let g x y = x + y;;
# val g : int -> int -> int = <fun>
g 2 3;;
# - : int = 5
g(2,3);; (* erreur *)
g(2,3);; (* erreur *)
 ^^^^^
Error: This expression has type 'a * 'b
       but an expression was expected of type int
```

Fonction qui rend deux résultats I

```
let rec split pivot liste = match liste with
  | [] -> ([], [])
  | h::r ->
      let (r1,r2) = split pivot r
      in if h<pivot then (h::r1,r2) else (r1,h::r2);;
# val split : 'a -> 'a list -> 'a list * 'a list = <fun>

split 17 [4;67;1;13;25];;
# - : int list * int list = ([4; 1; 13], [67; 25])
```

Comparer des tuples I

```
(1,2,3);;
# - : int * int * int = (1, 2, 3)
(1,(2,3));;
# - : int * (int * int) = (1, (2, 3))
(1,2,3) < (1,(2,3));;
(1,2,3) < (1,(2,3));;
          ^^^^^^^^^

Error: This expression has type 'a * 'b
       but an expression was expected of type
         int * int * int
```

Comparer des tuples II

```
(1,2,3) < [1;2;3];; (* pas confondre n-uplets et listes *)
(1,2,3) < [1;2;3];; (* pas confondre n-uplets et listes *)
          ^^^^^^^

Error: This expression has type 'a list
       but an expression was expected of type
         int * int * int
[1,2,3];;
# - : (int * int * int) list = [(1, 2, 3)]
(1,2,3);;
# - : int * int * int = (1, 2, 3)
1,2,3;;
# - : int * int * int = (1, 2, 3)
```

Listes d'association

- Liste de paires
- Très souvent utilisées pour représenter des fonctions à domaine fini.
- Il y a des fonctions utiles dans la bibliothèque `List`.

Exemple de liste d'association I

```
let bureaux = [ ("pierre", 101); ("julie", 103); ("paul", 205) ];;
# val bureaux : (string * int) list =
  [("pierre", 101); ("julie", 103); ("paul", 205)]
let rec trouver clef liste = match liste with
  | [] -> failwith ("Clef␣pas␣trouvee␣:"^clef)
  | (c,v)::r when c=clef -> v
  | _::r -> trouver clef r;;
# val trouver : string -> (string * 'a) list -> 'a = <fun>

trouver "julie" bureaux;;
# - : int = 103
trouver "marie" bureaux;;
# Exception: Failure "Clef␣pas␣trouvee␣:marie".
```

Exemple de liste d'association I

```
let alphabet = [ ("a",1); ("b",2); ("c",3)] ;;
# val alphabet : (string * int) list =
  [("a", 1); ("b", 2); ("c", 3)]
List.assoc "b" alphabet;;
# - : int = 2
List.assoc "toto" alphabet;;
# Exception: Not_found.
List.mem_assoc "a" alphabet;;
# - : bool = true
List.mem_assoc "titi" alphabet;;
# - : bool = false
```

Enregistrements

- Anglais : *record*
- C'est un n-uplet avec un nom où chaque champ a son propre nom et son propre type
- L'ordre des champs n'a pas d'importance (contrairement aux listes et aux n-uplets)
- Moins de modifications dans le code quand on ajoute un champ à un enregistrement, que quand on ajoute une composante à un n-uplet
- Pas besoin de connaître le nombre exact de champs pour accéder à un champ particulier d'un enregistrement.
- Fragile: ne pas utiliser le même nom de champ pour deux types différents

Exemple avec enregistrement I

```
(* declarer le type*)
type date = {
  day: int;
  month: string;
  year: int
};;
# type date = { day : int; month : string; year : int; }
(* definir un element de ce type*)
let today = {
  day = 7;
  month = "october";
  year = 2010;
};;
# val today : date =
  {day = 7; month = "october"; year = 2010}
```

Exemple avec enregistrement II

```
(* defintion  incomplete *)
let today = {
day = 7;
month = "october";
};;
# Lines 2-5, characters 12-1:
2 | ............{
3 | day = 7;
4 | month = "october";
5 | }..
Error: Some record fields are undefined: year
```

Exemple avec enregistrement III

```
(*acceder a un champ correct*)
today.year;;
# - : int = 2010
(*acceder a un champ erronne*)
today.annee;;
today.annee;;
      ^^^^^
Error: This expression has type date
       The field annee does not belong to type date
(*changer le champ d'un objet*)
let next_week = {today with day=14};;
# val next_week : date =
  {day = 14; month = "october"; year = 2010}
```

Exemple avec enregistrement IV

```
next_week.day;;
# - : int = 14
```

Un autre exemple avec enregistrement I

```
type r1 = {nom: string; age: int};;
# type r1 = { nom : string; age : int; }

let e1= {nom="Lucie"; age=17};;   (*ordre sans importance*)
# val e1 : r1 = {nom = "Lucie"; age = 17}
let e2 = {age=17; nom="Lucie"};; (*ordre sans importance*)
# val e2 : r1 = {nom = "Lucie"; age = 17}

e1.nom = e2.nom;;      (*egalite des champs*)
# - : bool = true
e1=e2;;                (*egalite des  enregistrements *)
# - : bool = true
```

Un autre exemple avec enregistrement II

```
(* projection alternative avec match explicite *)
let proj_nom elem = match elem with {nom;age} -> nom;;
# val proj_nom : r1 -> string = <fun>
proj_nom e1;;
# - : string = "Lucie"

(* projection alternative avec match implicite *)
let proj_nom {nom;age} = nom;;
# val proj_nom : r1 -> string = <fun>
proj_nom e1;;
# - : string = "Lucie"
```

Les types énumérés

- Nécessitent une définition de type
- Énumération des *constructeurs* de ce type
- Les constructeurs commencent obligatoirement sur une lettre en majuscule (ce que les distingue des identificateurs)
- Un constructeur ne peut appartenir qu'à un seul type
- Il s'agit d'un cas particulier des types *somme*

Type énuméré I

```
type couleur = Blanc | Noir | Vert | Rouge;;
# type couleur = Blanc | Noir | Vert | Rouge
let x = Blanc and y = Noir;;
# val x : couleur = Blanc
val y : couleur = Noir
x < y;;
# - : bool = true

type non_couleur = Blanc | Noir;; (* eviter la repetition des noms*)
# type non_couleur = Blanc | Noir
```

Type énuméré II

```
Blanc > Rouge;;
Blanc > Rouge;;
        ^^^^^
Error: This variant expression is expected to have type
         non_couleur
       The constructor Rouge does not belong to type non_couleur
```

Déconstruction des types énumérés I

```
type couleur = Blanc | Noir | Vert | Rouge;;
# type couleur = Blanc | Noir | Vert | Rouge
let inverse c = match c with
  | Blanc -> Noir
  | Noir -> Blanc
  | Vert -> Rouge
  | Rouge -> Vert;;
# val inverse : couleur -> couleur = <fun>
```

Les types algébriques

Trois généralisations des types énumérés :

1. Les constructeurs peuvent prendre un argument
2. Les types peuvent être polymorphes
3. Les types peuvent être récursifs

Syntaxe générale du type somme

```
type typename =
| Id1 of type1
| Id2 of type2
...
| Idn of typen;;
```

- Chaque identificateur `Idi` doit commencer par une lettre majuscule
- La partie `of typei` n'est pas obligatoire (cf type énuméré)

Les entiers I

```
type euros = Zero | Piece of float | Billet of int;;
# type euros = Zero | Piece of float | Billet of int

let pm1=Zero;;
# val pm1 : euros = Zero
let pm2=Piece(0.5);;
# val pm2 : euros = Piece 0.5

let convert euros = match euros with
|  Zero      -> 0.
|  Piece(x)  -> x
|  Billet(x) -> float_of_int x;;
# val convert : euros -> float = <fun>
```

Le type option et le type result I

```
(* Ce type est predefini en OCaml*)
type 'a option = | None | Some of 'a;;
# type 'a option = None | Some of 'a

let predecessor n =
 if n = 0 then None else Some (n - 1);;
# val predecessor : int -> int option = <fun>

(* Le type des resultats des fonctions partielles *)
type ('a, 'b) result =
| Ok of 'a     (* "Ok v" : le calcul s'est evalue normalement en "v". *)
| Error of 'b (* "Error e" : le calcul a echoue avec l'erreur "e". *);;
# type ('a, 'b) result = Ok of 'a | Error of 'b
```

Le type option et le type result II

```
let total_division a b =
  if b = 0 then Error "Division␣by␣zero" else Ok (a / b);;
# val total_division : int -> int -> (int, string) result =
  <fun>
```

Les cartes I

```
type couleur = Pique | Coeur | Carreau | Trefle;;
# type couleur = Pique | Coeur | Carreau | Trefle
type carte =
  | As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Numero of int * couleur;;
# type carte =
    As of couleur
  | Roi of couleur
  | Dame of couleur
  | Valet of couleur
  | Numero of int * couleur
```

Les cartes II

```
Dame Coeur;;
# - : carte = Dame Coeur
Numero (9, Trefle);;
# - : carte = Numero (9, Trefle)
let valeur atout carte =
  match carte with
    | As(_) -> 11
    | Roi(_) -> 4
    | Dame(_) -> 3
    | Valet(c) -> if c=atout then 20 else 2
    | Numero(10,_) -> 10
    | Numero(9,c) -> if c=atout then 14 else 0
    | Numero(_,_) -> 0;;
# val valeur : couleur -> carte -> int = <fun>
```

Les cartes III

```
valeur Trefle (Dame Coeur);;
# - : int = 3
valeur Trefle Dame(Coeur);; (* erreur de syntaxe *)
valeur Trefle Dame(Coeur);; (* erreur de syntaxe *)
^^^^^^
Error: This function has type couleur -> carte -> int
       It is applied to too many arguments;
       maybe you forgot a ';'.
valeur Trefle (Numero(9,Trefle));;
# - : int = 14
```

Types somme versus type produit

- Un type somme est une union disjointe des valeurs de ses composantes
  `type ts = C of color | D of day;;`
  contient $4 + 7 = 11$ valeurs.
- Un type produit est un produit cartésien
  `type tp = color * day;;`
  contient 4 x 7 = 28 valeurs

Types sommes polymorphes

- Le type est parametré par un type *polymorphe*, c'est à dire contient une variables de type de la forme `'a`
- Rappel: une variable de type dénote un type arbitraire

Type somme polymorphe simple (un seul cas) I

```
type ('a,'b) pair = Pair of 'a*'b;;
# type ('a, 'b) pair = Pair of 'a * 'b
let x = Pair(1, "toto");;
# val x : (int, string) pair = Pair (1, "toto")
let y = Pair(5.8, int_of_string);;
# val y : (float, string -> int) pair = Pair (5.8, <fun>)
x<y;; (* types differents ! *)
x<y;; (* types differents ! *)
  ^
Error: This expression has type (float, string -> int) pair
       but an expression was expected of type
         (int, string) pair
       Type float is not compatible with type int
```

Type somme polymorphe (deux cas) I

```
type 'a option = None | Some of 'a;;
# type 'a option = None | Some of 'a

let frac = function
  | _, 0  -> None
  | x , y -> Some ( x / y );;
# val frac : int * int -> int option = <fun>

3/0;;
# Exception: Division_by_zero.
frac (3 ,0);;
# - : int option = None
frac (3 ,1);;
# - : int option = Some 3
```

Type somme polymorphe (deux cas) I

```
type ( 'a , 'b ) twotypes = A of 'a | B of 'b;;
# type ('a, 'b) twotypes = A of 'a | B of 'b

[A 1; B "␣toto␣" ; A 3 ];;
# - : (int, string) twotypes list = [A 1; B "␣toto␣"; A 3]

let a = [A 1; A 3 ];;
# val a : (int, 'a) twotypes list = [A 1; A 3]

let b = [B "␣toto␣" ];;
# val b : ('a, string) twotypes list = [B "␣toto␣"]

a@b;;
# - : (int, string) twotypes list = [A 1; A 3; B "␣toto␣"]
```

Types somme récursifs

- La définition du type fait appel au type lui-même
- Il n'y a pas de `type rec`
- Utile seulement quand il y a un cas de base.
- Très puissant
- Cas typiques: listes, arbres, formules, arbres de syntaxe, etc

Les formules logiques I

```
type formula =
  | True
  | False
  | Var of string
  | Neg of formula
  | And of formula*formula
  | Or of formula*formula;;
# type formula =
    True
  | False
  | Var of string
  | Neg of formula
  | And of formula * formula
  | Or of formula * formula
```

Les formules logiques II

```
And(Var "x", Or (False, Neg (Var "z")));;
# - : formula = And (Var "x", Or (False, Neg (Var "z")))

let impl p q = Or(Neg p, q);;
# val impl : formula -> formula -> formula = <fun>

let rec taille p = match p with
  | True       -> 0
  | False      -> 0
  | Var _      -> 1
  | Neg p -> 1 + (taille p)
  | And(p1,p2) -> 1+(taille p1)+(taille p2)
  | Or(p1,p2)  -> 1+(taille p1)+(taille p2);;
# val taille : formula -> int = <fun>
```

Les formules logiques III

```
taille (And(Var "x", Or (Var "y", Neg (Var "z"))));;
# - : int = 6
```

Les expressions arithmétiques I

```
type expr =
  | Var of string
  | Add of expr * expr
  | Mult of expr * expr;;
# type expr =
    Var of string
  | Add of expr * expr
  | Mult of expr * expr
```

Les expressions arithmétiques II

```
let rec eval = fun environment expr -> match expr with
  | Var x -> List.assoc x environment
  | Add(e1,e2) -> (eval environment e1)
                    + (eval environment e2)
  | Mult(e1,e2) -> (eval environment e1)
                    * (eval environment e2);;
# val eval : (string * int) list -> expr -> int = <fun>

eval [("x",2); ("y",5)]
       (Add(Var "x", Mult(Var "x", Var "y")));;
# - : int = 12
```

La définition alternative des listes I

```
type 'a liste =
  | Nil
  | Cons of 'a * 'a liste;;
# type 'a liste = Nil | Cons of 'a * 'a liste
```

```
(* [42;17] *)
let l1 = Cons(42, Cons(17,Nil));;
# val l1 : int liste = Cons (42, Cons (17, Nil))
```

```
(* map f  liste *)
let rec mon_map f l = match l with
  | Nil       -> Nil
  | Cons(h,t) -> Cons(f h,mon_map f t);;
# val mon_map : ('a -> 'b) -> 'a liste -> 'b liste = <fun>
```

La définition alternative des listes II

```
mon_map (function x -> x+1) l1;;
# - : int liste = Cons (43, Cons (18, Nil))
```

Les arbres étiquétés I

```
type 'a bintree =
  | Vide
  | Node of 'a * 'a bintree * 'a bintree;;
# type 'a bintree =
    Vide
  | Node of 'a * 'a bintree * 'a bintree

let rec size x = match x with
  | Vide -> 0
  | Node(_,left,right) -> 1+(size left)+(size right);;
# val size : 'a bintree -> int = <fun>
```

Les arbres étiquétés II

```
size (Node(2,
           Vide,
           Node(42, Vide, Vide)));;
# - : int = 2

size (Node("a",
           Node("b", Vide, Vide),
           Node("c", Vide, Vide)));;
# - : int = 3
```