Programmation Fonctionnelle
Cours 04

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

3 octobre 2016

Les types algebriques

ou comment construire ses propres types

```
(* produit *)
type point = float * float

(* somme *)
type number = Zero | Integer of int | Real of float

(* recursion *)
type unary = Zero | Succ of unary

(* polymorphisme *)
type 'a option = None | Some of 'a
```

Type produit ou *n*-uplets

```
type point = float * float
```

Construire *n*-uplets

- Les produits cartésiens sont une parmi les constructions les plus simples de structures de données.

```
# let p = 1, "Hello";;
val p : int * string = (1, "Hello")
```

- On les construit en combinant les éléments par la virgule (souvent écrit entre parenthèses, mais pas nécessaire):

```
# 3.4, (fun x y -> x*y), 'a';;
- : float * (int -> int -> int) * char = (3.4, <fun>, 'a')
```

- Une valuer du type produit est une *n*-uplet de valuers des types correspondants:

```
# "Hello"^" world",(fun x ->x*x)3;;
- : string * int = ("Hello world", 9)
```

Construire *n*-uplets

- Les produits cartésiens sont une parmi les constructions les plus simples de structures de données.

```
# let p = 1, "Hello";;
val p : int * string = (1, "Hello")
```

- On les construit en combinant les éléments par la virgule (souvent écrit entre parenthèses, mais pas nécessaire):

```
# 3.4, (fun x y -> x*y), 'a';;
- : float * (int -> int -> int) * char = (3.4, <fun>, 'a'
```

- Une valuer du type produit est une *n*-uplet de valuers des types correspondants:

```
# "Hello"^" world",(fun x ->x*x)3;;
- : string * int = ("Hello world", 9)
```

Construire *n*-uplets

- Les produits cartésiens sont une parmi les constructions les plus simples de structures de données.

```
# let p = 1, "Hello";;
val p : int * string = (1, "Hello")
```

- On les construit en combinant les éléments par la virgule (souvent écrit entre parenthèses, mais pas nécessaire):

```
# 3.4, (fun x y -> x*y), 'a';;
- : float * (int -> int -> int) * char = (3.4, <fun>, 'a'
```

- Une valuer du type produit est une *n*-uplet de valuers des types correspondants:

```
# "Hello"^"␣world",(fun x ->x*x)3;;
- : string * int = ("Hello␣world", 9)
```

Un produit n'est pas une liste

- Attention: à ne pas confondre les produits avec les listes !

```
# let pair = 1,2;;
val pair : int * int = (1, 2)
# let liste = [1;2];;
val liste : int list = [1; 2]
# List.hd pair;;
          ^^^
Error: This expression has type int * int but
an expression was expected of type 'a list
```

- Les listes sont séquences *de longueur variable* et des éléments *du même type*

```
# 1,"Hello";;
- : int * string = (1, "Hello")
# [1;"Hello"];;
      ^^^^
Error: This expression has type string but an
expression was expected of type int
```

- Les *n*-uplets sont séquences dont la longueur est fixée par le type et les éléments peuvent être hétérogènes

Un produit n'est pas une liste

- Attention: à ne pas confondre les produits avec les listes !

```
# let pair = 1,2;;
val pair : int * int = (1, 2)
# let liste = [1;2];;
val liste : int list = [1; 2]
# List.hd pair;;
          ^^^
Error: This expression has type int * int but
an expression was expected of type 'a list
```

- Les listes sont séquences *de longueur variable* et des éléments *du même type*

```
# 1,"Hello";;
- : int * string = (1, "Hello")
# [1;"Hello"];;
       ^^^^
Error: This expression has type string but an
expression was expected of type int
```

- Les *n*-uplets sont séquences dont la longueur est fixée par le type et les éléments peuvent être hétérogènes

Un produit n'est pas une liste

- Attention: à ne pas confondre les produits avec les listes !

```
# let pair = 1,2;;
val pair : int * int = (1, 2)
# let liste = [1;2];;
val liste : int list = [1; 2]
# List.hd pair;;
          ^^^
Error: This expression has type int * int but
an expression was expected of type 'a list
```

- Les listes sont séquences *de longueur variable* et des éléments *du même type*

```
# 1,"Hello";;
- : int * string = (1, "Hello")
# [1;"Hello"];;
       ^^^^
Error: This expression has type string but an
expression was expected of type int
```

- Les *n*-uplets sont séquences dont la longueur est fixée par le type et les éléments peuvent être hétérogènes

Deconstruire *n*-uplets

Pour déconstruire les *n*-uplets, on peut

- utiliser les projections, dans le cas binaires:

```
# fst p;;
- : int = 1

# snd p;;
- : string = "Hello"
```

- ou, dans le cas général, le filtrage par motif:

```
# let trd triplet = match triplet with (x,y,z) -> z;;
val trd : 'a * 'b * 'c -> 'c = <fun>

# trd (1,2,3);;
- : int = 3
```

- ou le let (raccourci pour le filtrage à un seul cas):

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

Deconstruire *n*-uplets

Pour déconstruire les *n*-uplets, on peut

- utiliser les projections, dans le cas binaires:

```
# fst p;;
- : int = 1

# snd p;;
- : string = "Hello"
```

- ou, dans le cas général, le filtrage par motif:

```
# let trd triplet = match triplet with (x,y,z) -> z;;
val trd : 'a * 'b * 'c -> 'c = <fun>

# trd (1,2,3);;
- : int = 3
```

- ou le let (raccourci pour le filtrage à un seul cas):

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

Deconstruire *n*-uplets

Pour déconstruire les *n*-uplets, on peut

- utiliser les projections, dans le cas binaires:

```
# fst p;;
- : int = 1

# snd p;;
- : string = "Hello"
```

- ou, dans le cas général, le filtrage par motif:

```
# let trd triplet = match triplet with (x,y,z) -> z;;
val trd : 'a * 'b * 'c -> 'c = <fun>

# trd (1,2,3);;
- : int = 3
```

- ou le let (raccourci pour le filtrage à un seul cas):

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

Ça sert à quoi les *n*-uplets ?

- Écrire des fonctions qui envoient *n*-uplets de valuers en sortie:

```
# 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;6;19; 0; 3049];;]
- : int list * int list = ([4; 6; 0], [19; 3049])
```

- Écrire fonctions qui prennent *n*-uplets de valeurs en argument (Attention : pas confondre avec fonctions plusieurs arguments)

```
# let add_pair (x,y) =x+y;;
val add_pair : int * int -> int = <fun>

# let add x y = x+y;;
val add : int -> int -> int = <fun>
```

- Construire des types plus complexes:
  - les listes d'association
  - les types algébriques

Ça sert à quoi les *n*-uplets ?

- Écrire des fonctions qui envoient *n*-uplets de valuers en sortie:

```
# 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;6;19; 0; 3049];;]
- : int list * int list = ([4; 6; 0], [19; 3049])
```

- Écrire fonctions qui prennent *n*-uplets de valeurs en argument (Attention : pas confondre avec fonctions plusieurs arguments)

```
# let add_pair (x,y) =x+y;;
val add_pair : int * int -> int = <fun>

# let add x y = x+y;;
val add : int -> int -> int = <fun>
```

- Construire des types plus complexes:
  - les listes d'association
  - les types algébriques

Ça sert à quoi les *n*-uplets ?

- Écrire des fonctions qui envoient *n*-uplets de valuers en sortie:

```
# 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;6;19; 0; 3049];;]
- : int list * int list = ([4; 6; 0], [19; 3049])
```

- Écrire fonctions qui prennent *n*-uplets de valeurs en argument (Attention : pas confondre avec fonctions plusieurs arguments)

```
# let add_pair (x,y) =x+y;;
val add_pair : int * int -> int = <fun>

# let add x y = x+y;;
val add : int -> int -> int = <fun>
```

- Construire des types plus complexes:
  - les listes d'association
  - les types algébriques

Type produit (Exercice)

On définit un polygone comme une liste de paires de float: `type polygone = (float*float) list`. Par exemple un carré de coté 1 et avec un sommet sur l'origine peut être représenté par `[(0.,0.); (1.,0.); (1.,1.); (0.,1.)]`. Définir alors la fonction `perim : polygone -> float` qui renvoie le périmètre d'un polygone donné en entrée.

On rappelle que la distance entre deux points (a, b), (c, d) *est donnée par:* $\sqrt{(a-c)^2 + (b-d)^2}$.

Type somme

```
type number = Zero | Integer of int | Real of float
```

Type somme

```
type typename =
  | Identifier1 of type1
  | Identifier2 of type2
  ....
  | Identifiern of typen
```

- défini par un ensemble de cas séparés par une barre | en utilisant le mot clé `type`
  (première barre | optionnelle)
- chaque cas caractérisé par un constructeur
  l'identificateur doit commencer par une lettre majuscule
- chaque constructeur peut (pas obligatoire) avoir un argument d'un certain type.

Type somme

```
# type number =
    | Zero
    | Integer of int
    | Real of float;;
type number = Zero | Integer of int | Real of float
```

- Une valeur est un constructeur appliqué à une valeur de l'argument si besoin

```
# let x = Zero;;
val x : number = Zero

# let y = Real (3./.2.5);;
val y : number = Real 1.2
```

- Les fonctions peuvent être définies par filtrage par motif

```
# let float_of_number = function
    | Zero -> 0.0
    | Integer x -> float_of_int x
    | Real x -> x;;
val float_of_number : number -> float = <fun>
```

Type somme

```
# type number =
    | Zero
    | Integer of int
    | Real of float;;
type number = Zero | Integer of int | Real of float
```

- Une valeur est un constructeur appliqué à une valeur de l'argument si besoin

```
# let x = Zero;;
val x : number = Zero

# let y = Real (3./.2.5);;
val y : number = Real 1.2
```

- Les fonctions peuvent être définies par filtrage par motif

```
# let float_of_number = function
    | Zero -> 0.0
    | Integer x -> float_of_int x
    | Real x -> x;;
val float_of_number : number -> float = <fun>
```

Type somme

```
# type number =
    | Zero
    | Integer of int
    | Real of float;;
type number = Zero | Integer of int | Real of float
```

- Une valeur est un constructeur appliqué à une valeur de l'argument si besoin

```
# let x = Zero;;
val x : number = Zero

# let y = Real (3./.2.5);;
val y : number = Real 1.2
```

- Les fonctions peuvent être définies par filtrage par motif

```
# let float_of_number = function
    | Zero -> 0.0
    | Integer x -> float_of_int x
    | Real x -> x;;
val float_of_number : number -> float = <fun>
```

Type somme

- les types énumérés sont un cas special de type somme (où les constructeurs n'ont pas d'arguments):

```
type color = Green | Blue | White | Black
type day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

- Attention à ne pas confondre type somme et type produit:
  - un type somme est une union disjointe des valeurs de ses composants, lorsque un type produit est un produit cartésien
  - le type:

    ```
    type tp = color * day
    ```

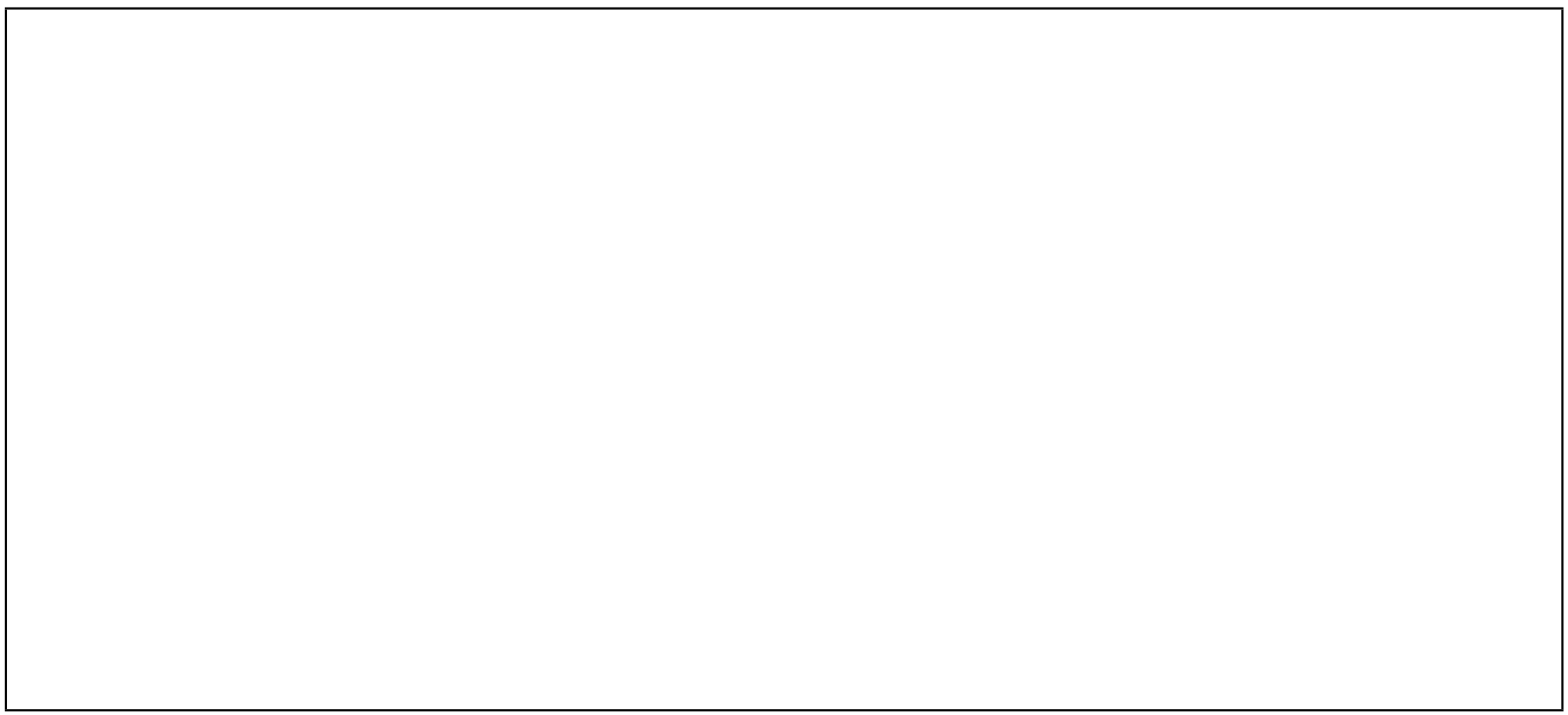
    contient $4 \times 7 = 28$ valeurs
  - le type:

    ```
    type ts = C of color | D of day;;
    ```

    contient $4 + 7 = 11$ valeurs.

Type somme (Exercice)

- État donné le type `number = Zero | Integer of int | Real of float`, définir l'opération d'addition effectuant une conversion implicite d'un `Integer` en `Real` lorsque le premier est additionné à un `Real` ("comme en Java").



Type récursif

```
type unary = Z | S of unary
```

Type récursif

- La définition du type permet des appels récursifs:

```
# type unary =
    | Z
    | S of unary
type unary = Z | S of unary

# let trois = S (S (S Z));;
val trois : unary = S (S (S Z))
```

- Ceci donne des structures de données avec une infinité de valeurs

```
Z,  S Z,  S (S Z),  S (S (S Z)),  S (S (S (S Z))),  ...
```

- Attention à ne pas mettre le mot clé **rec**:

```
# type rec unary =  Z | S of unary
        ^^^
Error: Syntax error
```

Type récursif (Exercice)

- Définir la fonction `unary_of_int : int -> unary`.

- Définir l'addition `add : unary -> unary -> unary`

Type polymorphe

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

Type polymorphe

- La définition du type permet aussi des variables de type:

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

# Some 3;;
- : int option = Some 3

# 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
```

Type polymorphe

- on peut avoir même plusieurs variables de type:

```
# 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"]
```

- plus sur les types polymorphes dans la suite du cours...

Types algebriques

- un type algébrique est un type construit par les types de bases, les variables de type, les sommes, les produits, la récursion.
- c'est une façon extrêmement puissante et simple pour construire des structures de données, comme
  - les listes,
  - les arbres,
  - les graphes,
  - les formules (logiques, arithmétiques, etc...)
  - les arbres de syntaxe abstraite
  - etc...

Les propositions logiques

```
1  (*boolean formulas*)
2  type prop =
3      | True
4      | False
5      | Var of string
6      | Neg of prop
7      | And of prop*prop
8      | Or of prop*prop
9
10 (* (non x) et (Vrai ou y) *)
11 let p1 = And (Neg (Var "x") , Or (True, Var "y"))
12
13 (*implication map: p,q -> (non p) ou q*)
14 let impl p q = Or (Neg p, q)
```

Les listes (fait maison !)

Les listes (fait maison !)

```
1  (* listes *)
2  type 'a list_maison =
3      | Nil
4      | Cons of 'a * ('a listes)
5
6  (* [1; 2; 3] *)
7  let onetwothree = Cons (1, Cons (2, Cons (3, Nil)))
8
9  (*map fait maison*)
10 let rec map_maison f l = match l with
11       Nil -> Nil
12     | Cons (x,tl) -> Cons ((f x),(map_maison f tl))
```

Les arbres étiquetés

Les arbres étiquetés

```
1  (*binary labelled trees*)
2  type 'a tree = Nil
3         | Node of ('a * 'a tree * 'a tree)
4
5  (*the empty tree, of type 'a tree*)
6  let t1 = Nil ;;
7
8  (*an int tree*)
9  let t2 = Node (1, Node (2,Nil,Nil), Nil);;
10
11 (*a string tree*)
12 let t2 = Node ("root",
13           Node ("gauche", Nil, Nil),
14           Node ("droite", Nil, Nil))
```

Arbres de syntaxes

```
1  (**abstract syntax tree simplifie depuis cours de compilation M1*)
2  type program = definition list
3
4  and definition =
5    | DefineValue of pattern * expression
6
7  and expression =
8    | Int of int
9    | Variable of identifier
10   | Apply of expression * expression
11   | Fun of identifier * expression
12   | IfThenElse of expression * expression * expression
13   | Match of expression * (pattern * expression list)
14
15 and pattern =
16   | PVariable     of identifier
17   | PTuple        of identifier list
18   | PTaggedValues of tag * identifier list
19
20 and tag =
21   | Constructor of string
22
23 and identifier =
24   | Id of string
```