Types Algébriques

Alan Schmitt

19 septembre 2018

Introduction

Types

Dans les langages de programmation un type est un ensemble de valeurs possibles
Dire « l'expression e est de type `T` », signifie « les valeurs que de l'évaluation de e appartiennent à `T` »

Inférence de types

- OCaml n'impose pas de déclarer le type des expressions ou des noms
- OCaml, au vu d'une expression, sait, sans ambiguïté, trouver le domaine auquel elle appartient

C'est l'inférence de type:

```
let _ = 5 + 2
```

```
- : int = 7
```

```
let _ = fun () -> 5 = 2
```

```
- : unit -> bool = <fun>
```

Type produit et type somme

Les domaines prédéfinis ne suffisent pas toujours, et le programmeur peut alors définir de nouveaux domaines de valeurs.

1. définir un type T comme un produit de types plus simples, comme
$$T = \{(i, j) | i \in T1 \wedge j \in T2\}$$
Les valeurs de `T` sont des couples de valeurs prises dans $T1$ et $T2$.
2. définir un type comme une somme (énumération de valeurs), comme
$$T = \{v1, v2, ..., vn\}$$
les valeurs de T sont explicitement listées.

Type Produit

Nuplets

Type produit anonyme

```
let _ = (4,true,1.0)
```

```
- : int * bool * float = (4, true, 1.)
```

Utile pour retourner plusieurs valeurs

```
let quotient_reste m n =
  let quotient = m / n in
  let reste = m mod n in
  (quotient,reste)
```

```
val quotient_reste : int -> int -> int * int = <fun>
```

```
let (q,r) = quotient_reste 13 5
```

```
val q : int = 2
val r : int = 3
```


Enregistrement

Type produits nommés

```
type t = {i : t1; j : t2}
```

Exemple:

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

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

```
let p1 = {x = 1.; y = 2. }
```

```
val p1 : point = {x = 1.; y = 2.}
```

```
let _ = p1.x
```

```
- : float = 1.
```

Type produit : filtrage

```
type cercle = {centre : point; rayon : float}
```

```
type cercle = { centre : point; rayon : float; }
```

Les enregistrement permettent d'accéder aux champs par filtrage

```
let symdiag {x=a ; y=b} = {x=b ; y=a}
```

```
val symdiag : point -> point = <fun>
```

ou par leur nom

```
let symdiag p = {x=p.y ; y=p.x}
```

```
val symdiag : point -> point = <fun>
```


Exercice : type produit

Définir un prédicat qui rend vrai si un cercle a son centre sur l'axe des y.

```
let ycercle c = match c with
| {centre = {x=0.; y =_}; rayon = _} -> true
| _                                  -> false
```

```
val ycercle : cercle -> bool = <fun>
```

Définir un prédicat qui rend vrai si un cercle a son centre sur la diagonale principale

```
let diagcercle {centre = {x=x;y=y}; rayon = _} = x=y
```

```
val diagcercle : cercle -> bool = <fun>
```

On peut omettre des champs dans le filtrage

```
let diagcercle {centre = {x=x;y=y}} = x=y
```

```
val diagcercle : cercle -> bool = <fun>
```

Nuplets ou enregistrements ?

Les nuplets ne requièrent pas de déclaration préalable, mais

```
let _ = (4.,2.)
```

```
- : float * float = (4., 2.)
```

peut représenter le point `(4.,2.)` du plan, le nombre complexe en polaire $4e^{2i\pi}$, un couple de mesures ... Les enregistrements sont plus explicites.

```
type polar = { rho : float; theta : float }
```

```
type polar = { rho : float; theta : float; }
```


Avantages des enregistrements

- accès par nom au champ d'un enregistrement

```
let _ = p1.x
```

```
- : float = 1.
```

- pas d'ordre fixé

```
let p2 = { y = 2.; x = 4. }
```

```
val p2 : point = {x = 4.; y = 2.}
```

- création d'un nouvel enregistrement à partir d'un ancien

```
let p3 = { p1 with x = 5. }
```

```
val p3 : point = {x = 5.; y = 2.}
```

Type Somme

Type Somme

$$T = \{v1, v2, ..., vn\}$$

```
type t = V1 | V2 | ... | Vn
```

Type Somme : exemple

Un exemple concret :

```
type direction  = Nord | Sud | Est | Ouest
```

```
type direction = Nord | Sud | Est | Ouest
```

Chaque élément du type est un constructeur

```
let d = Nord
```

```
val d : direction = Nord
```

On utilise le filtrage pour inspecter ces valeurs

```
let agauche d = match d with
| Nord -> Ouest
| Ouest -> Sud
| Sud -> Est
| Est -> Nord
```

```
val agauche : direction -> direction = <fun>
```


Les types OCaml sont disjoints

On ne peut pas déclarer le même constructeur dans plusieurs types

```
type estouest  = Est | Ouest (* ceci masque l'ancienne déclaration *)
let res = Est
```

```
type estouest = Est | Ouest
val res : estouest = Est
```

On ne peut pas utiliser des valeur ou des types comme constructeurs

```
type direction = 1 | 2 | 3 | 4
```

```
Characters 17-18:
  type direction = 1 | 2 | 3 | 4;;
                   ^

Error: Syntax error
```

```
type personne = int | string
```

```
Characters 20-21:
  type personne = int | string;;
                        ^

Error: Syntax error
```

Type somme avec argument

```
type personne =
| Nom of string
| Num of int
```

```
type personne = Nom of string | Num of int
```

nous avons un domaine « personne » dont les valeurs sont des chaînes ou des numéros qui permettent d'identifier des gens.
`Nom` et `Num` sont des constructeurs de valeur

```
let _ = Nom "Alan"
```

```
- : personne = Nom "Alan"
```

Ce ne sont pas des fonctions

```
let _ = Nom
```

```
Characters 8-11:
  let _ = Nom;;
          ^^^
Error: The constructor Nom expects 1 argument(s),
       but is applied here to 0 argument(s)
```


Type somme : filtrage

```
let _ = Nom "cesar"
```

```
- : personne = Nom "cesar"
```

```
let _ = Num 1480735186327
```

```
- : personne = Num 1480735186327
```

ces valeurs peuvent être filtrées dans les abstractions

```
let string_of_name n = match n with
| (Nom s) -> s
| (Num n) -> string_of_int n
```

```
val string_of_name : personne -> string = <fun>
```

Type somme et lisibilité des valeurs

Un cas particulier (un seul constructeur !)

```
type vitesse = KmParH of float
```

```
type vitesse = KmParH of float
```

il peut paraître lourd de manipuler `KmParH 70.4` plutôt que `70.4`, mais la valeur `KmParH 70.4` est plus explicite que `70.4` qui peut être une température. Cela documente également l'unité utilisée. De plus l'inférence de type peut faire des vérifications de cohérence. On ne pourra pas transmettre une température à une fonction qui attend une vitesse, ou se tromper d'unité.

Exercice : type somme (1)

Exemple :

```
type couleur = Trefle | Carreau | Coeur | Pique
type hauteur = As | Roi | Dame | Valet
             | Dix | Neuf | Huit | Sept
type carte   = Carte of  hauteur * couleur
```

```
type couleur = Trefle | Carreau | Coeur | Pique
type hauteur = As | Roi | Dame | Valet | Dix | Neuf | Huit | Sept
type carte = Carte of hauteur * couleur
```

Comment notez-vous le roi de pique ? Le 7 de trèfle ?
Définir une fonction qui teste si une carte est un Valet.
Définir une fonction qui associe à toute carte sa couleur « rouge » ou « noir »

Exercice : type somme (2)

```
let _ = Carte (Roi,Pique)
```

```
- : carte = Carte (Roi, Pique)
```

```
let _ = Carte (Sept,Trefle)
```

```
- : carte = Carte (Sept, Trefle)
```

```
let estvalet c = match c with
| Carte(Valet,_) -> true
| _ -> false
```

```
val estvalet : carte -> bool = <fun>
```

```
let coul c = match c with
| Carte (_,(Carreau | Coeur)) -> "rouge"
| _ -> "noir"
```

```
val coul : carte -> string = <fun>
```


Bilan

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

(* filtrage *)      {x = a ; y = 17} -> ...
(* construction *) {x = a ; y = a+2}
(* acces *)         point.x
```

```
type direction  = Nord | Sud | Est | Ouest

(* filtrage *)      | Nord -> ...
                    | Sud -> ...
                    | Est -> ...
                    | Ouest -> ...
(* construction *) Sud
```

```
type comp  =
| Pol of float * float
| Rect of float * float

(* filtrage *)      | Pol (m,a) -> ...
                    | Rect (0,i) -> ...
                    | Rect (x,i) -> ...
(* construction *) Pol(4+v,a)
```

Les Listes

Listes d'entiers

Les définitions de type somme peuvent être récursives, c'est à dire être utilisées dans leur propre définition. Exemple : les listes.

```
type intliste = Lvide | Cons of int * intliste
```

```
type intliste = Lvide | Cons of int * intliste
```

```
let _ = Cons(4,Cons(5,Lvide))
```

```
- : intliste = Cons (4, Cons (5, Lvide))
```

```
let rec lg l = match l with
| Lvide -> 0
| Cons(_, reste) -> 1 + lg reste
```

```
val lg : intliste -> int = <fun>
```

Listes et récursion

Un type récursif possède des cas de base (`Lvide`) et des cas récursifs (`Cons`). Un fonction récursive pour un tel type teste dans quel cas on est (avec un `match`) et s'appelle récursivement dans le cas récursif.

```
let rec funrec l = match l with
| Lvide -> cas_de_base
| Cons(tete, queue) -> ... funrec queue ...
```


Liste polymorphes

De nombreuses opérations sur les listes ne dépendent pas de la nature des valeurs dans la liste.
On peut alors définir un type de liste plus général

```
type 'a liste =
| Lvide
| Cons of 'a * 'a liste
```

```
type 'a liste = Lvide | Cons of 'a * 'a liste
```

Les listes en Ocaml

- Lvide $\longrightarrow$ []
- Cons $\longrightarrow$ `::` (en position infixe)

```
let _ = 4::5::[]
```

```
- : int list = [4; 5]
```

```
let rec lg l = match l with
| [] -> 0
| a::reste -> 1+ lg reste
```

```
val lg : 'a list -> int = <fun>
```


Notation pour les listes

`::` est utilisé en

- analyse lors du filtrage: `a :: reste` (on nomme a l'élément de tête et `reste` le reste de la liste argument)
- création de liste: `4 :: (5 :: [])`

On peut aussi écrire `[4; 5]` pour créer cette liste.
Attention, le constructeur `::` attend un élément à sa gauche et une liste à sa droite, et retourne une liste

```
let cons x y = x :: y
```

```
val cons : 'a -> 'a list -> 'a list = <fun>
```

On ne peut pas l'utiliser pour concaténer des listes:

```
let _ = [4] :: [5]
```

```
Characters 16-17:
  let _ = [4] :: [5];;
                  ^

Error: This expression has type int
       but an expression was expected of type int list
```

Fonctions sur les listes

Pour utiliser les fonctions de base sur les listes, fournies par Objective Caml dans le module «List» de sa bibliothèque standard on peut:

- utiliser le nom du module comme préfixe

```
List.length
List.append
```

- ouvrir le module List : toutes les fonctions qu'il contient deviennent alors directement accessibles

```
open List
length
append
```


Récursivité et listes

```
let rec f l = match l with
  | [] -> base                  (* cas de base *)
  | hd :: tl ->
    let _ = f1 hd in            (* avant l'appel récursif *)
    let res = f tl in           (* appel récursif *)
    let final = f2 hd res in    (* après l'appel récursif *)
    final
```

Sur la liste `[1; 2; 3]`, la séquence d'appels est la suivante:

```
f [1; 2; 3]
  f1 1
  f [2; 3] (* appel récursif *)
    f1 2
    f [3]   (* appel récursif *)
      f1 3
        f [] (* retourne le cas de base *)
      f2 3 base (* retourne res3 *)
    f2 2 res3 (* retourne res32 *)
  f2 1 res32 (* retourne res321, valeur finale *)
```

Traitements génériques `List.map`

Exemple 1:
map : faire subir le même traitement t à tous les éléments d'une liste l et produire une nouvelle liste

```
let rec map t l = match l with
| [] -> []
| (a::reste) ->
  let a' = t a in
  let reste' = map t reste in
  a':: reste'
```

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Cette fonction est fournie : `List.map`

Traitements génériques `List.fold_left`

Exemple 2 :
`list_it` : « cumuler » grâce à op tous les éléments d'une liste `l`.
Le cumul commence à e

```
list_it op e [x; y; ...z]  =
  op (... (op (op e x) y) ... )z
```

```
let rec list_it op e l = match l with
| [] -> e
| (a::reste) ->
  let e' = op e a in
  list_it op e' reste
```

```
val list_it : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Cette fonction est fournie : `List.fold_left`. Le cumul est fait en descendant dans les niveaux de récursivité

Traitements génériques `List.fold_right`

Exemple 3 :
`it_list` : « cumuler » grâce à op tous les éléments d'une liste `l`.
Le cumul commence à `e`

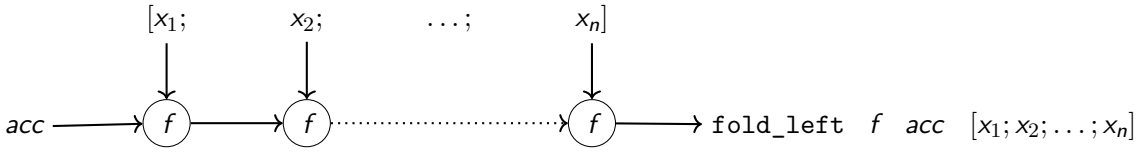
```
it_list op [x; y; ...z]  e =
  op x (op y ( ...(op z e) ...))
```

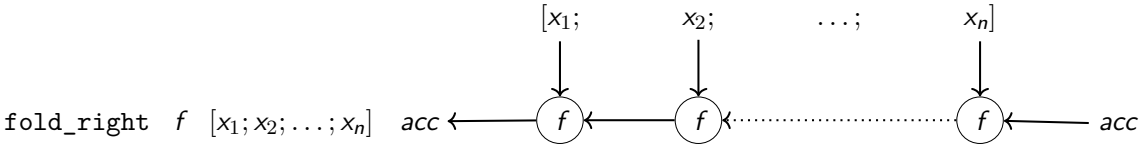
```
let rec it_list op l e =
 match l with
 |  [] ->  e
 | (a::reste) ->
   let e' = it_list op reste e in
   op a e'
```

```
val it_list : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Cette fonction est fournie : `List.fold_right`. Le cumul est fait en remontant dans les niveaux de récursivité

`List.fold_left` vs `List.fold_right`





List.for_all, List.exists, List.filter

Quelques fonctions du module List utiles pour les exercices suivants :

```
let _ = List.for_all (* for_all pred l *)
```

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

```
let _ = List.exists  (* exists pred l *)
```

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

```
let _ = List.filter  (* filter pred l *)
```

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


Exercices : fonctionnelles listes (1)

Tous les éléments de `l1` sont-ils dans `l2` ?

```
let inclus l1 l2 = ...
```

```
val inclus : 'a list -> 'a list -> bool = <fun>
```

```
let _ = inclus [1; 4; 3; 1] [7; 3; 1; 4]
```

```
- : bool = true
```

```
let inclus l1 l2 =
 List.for_all (fun x -> List.exists (fun y -> x=y) l2) l1
```

Exercices : fonctionnelles listes (2)

La liste `l` privée, de la première occurrence de `e`.

```
let sans e l = ...
```

```
val sans : 'a -> 'a list -> 'a list = <fun>
```

```
let _ = sans 2 [1;2;3;2;8;2;5]
```

```
- : int list = [1; 3; 2; 8; 2; 5]
```

```
let rec sans e l = match l with
| [] -> []
| a::r -> if a = e then r else a::(sans e r)
```


Exercices : fonctionnelles listes (3)

La liste `l` a-t-elle des éléments présents plus d'une fois ?

```
let sansd l = ...
```

```
val sansd : 'a list -> bool = <fun>
```

```
let _ = sansd [1;2;3;2;8;2;5]
```

```
- : bool = false
```

```
let sansd l =
 List.for_all (fun x -> List.for_all (fun y -> x<>y) (sans x l)) l
```

Exercices : fonctionnelles listes (4)

La liste `l` privée des éléments présents plus d'une fois.

```
let mksd l = ...
```

```
val mksd : 'a list -> 'a list = <fun>
```

```
let _ = mksd [1;2;3;2;8;2;5]
```

```
- : int list = [1; 3; 8; 5]
```

```
let mksd l =
 List.filter
  (fun x -> List.for_all (fun y -> x<>y) (sans x l))
  l
```


Concaténer et aplatir

Concaténer 2 listes : (@)

```
let rec concat l1 l2 =
  match l1 with
  | []    ->  l2
  | x::l  -> x :: concat l l2
```

```
val concat : 'a list -> 'a list -> 'a list = <fun>
```

Aplatir une liste de listes

```
let rec aplatir l = match l with
| [] -> []
| x::l' -> concat x (aplatir l')
```

```
val aplatir : 'a list list -> 'a list = <fun>
```

Triangle de Pascal (1)

Définir une fonction
`val genlist : 'a -> ('a -> 'a) -> int -> 'a list = <fun>`

telle que `genlist x suiv n` construit une liste de n éléments. Le premier de la liste est `x`, le second `suiv x`, etc ...

```
let rec genlist i suiv l = match l with
| 0 -> [ ]
| n -> i :: genlist (suiv i) suiv (n-1)
```

En utilisant genlist définir la fonction
`val fromto : int -> int -> int list = <fun>`

telle que `fromto a b` construit la liste des entiers de l'intervalle `[a,b]`

```
let fromto n m = genlist n succ (m-n+1)
```


Triangle de Pascal (2)

Définir une fonction
`val ligsuiv : int list -> int list = <fun>`

telle que `ligsuiv l` produit la ligne qui suit la ligne `l` dans le triangle de Pascal.

```
let ligsuiv l =
  let rec lsuiv l' = match l' with
   | [ ] -> [ ]
   | [x]   ->  [x]
   | (x::y::r) -> x+y :: lsuiv (y::r)
in 1::lsuiv l
```

Triangle de Pascal (3)

En utilisant genlist et ligsuiv définir la fonction
`val triangle : int -> int list list = <fun>`

telle que `triangle n` produit la liste des n premières lignes du triangle de Pascal

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
```

```
let triangle n = genlist [1] ligsuiv n
```

```
let _ = triangle 5
```

```
- : int list list =
[[1]; [1; 1]; [1; 2; 1]; [1; 3; 3; 1]; [1; 4; 6; 4; 1]]
```


Triangle de Pascal (alternative)

Chaque ligne commence par 1.

```
let rec ligsuiv l = match l with
| [] -> []
| [x]    ->  [x]
| (x::y::r) -> x+y :: ligsuiv (y::r)
```

```
val ligsuiv : int list -> int list = <fun>
```

```
let triangle n = genlist [1] (fun l -> 1 :: ligsuiv l) n
```

```
val triangle : int -> int list list = <fun>
```

Split and merge (1)

Le principe de ce tri :
Pour trier la liste `l`

- on scinde `l` en deux listes `l1` et `l2`
- on trie `l1`, donnant `lt1`
- on trie `l2`, donnant `lt2`
- on fusionne `lt1` et `lt2`

Définir les fonctions trie, scinde et fusionne

Split and merge (2)

La fonction `scinde` répartit les éléments d'une liste et rend un doublet de listes

```
let rec scinde l = match l with
| []  -> ([],[])
| [e] -> ([e],[])
| e::f::l -> let (l1,l2) = scinde l in
             (e::l1,f::l2)
```

```
val scinde : 'a list -> 'a list * 'a list = <fun>
```

Split and merge (3)

La fonction `fus` fusionne les éléments de deux listes triées et rend un liste triée

```
let rec fus lp = match lp with
| ([],l) -> l
| (l,[]) -> l
| (x::l1,y::l2) -> if x < y then  x :: fus(l1,y::l2)
                           else  y :: fus (x::l1,l2)
```

```
val fus : 'a list * 'a list -> 'a list = <fun>
```


Split and merge (4)

La fonction principale :

- on scinde `l` en deux listes `l1` et `l2`
- on trie `l1`, donnant `lt1`
- on trie `l2`, donnant `lt2`
- on fusionne `lt1` et `lt2`

```
let rec sam l = match l with
| [] -> []
| [e] -> [e]
| l  -> let (l1,l2) = scinde l in
        fus (sam l1, sam l2)
```

```
val sam : 'a list -> 'a list = <fun>
```

Split and merge (5)

La fonction de tri obtenue est générique

```
let _ = sam [4;2;7;5;1;7;3;9]
```

```
- : int list = [1; 2; 3; 4; 5; 7; 7; 9]
```

```
let _ = sam[(false,4);(true,2);(true,7);(false,5); (true,1);(false,7)]
```

```
- : (bool * int) list =
[(false, 4); (false, 5); (false, 7); (true, 1); (true, 2); (true, 7)]
```