# Types en OCaml

Evaluation d'une expression $\Longrightarrow$ type et valeur

```
# 10 + 4;;
- : int = 14
```

Un type peut-être vu un ensemble (finis ou infini) de valeurs.

Exemples :

type Booléen `bool = { true, false }`
type Caractères `char = { 'a', 'A', ...}`

Les fonctions OCaML sont typées: elles s'appliquent à des arguments ayant chacun un type défini et retournent une valeur d'un type également défini.

Un type avec un ensemble d'opérateurs s'appliquant sur les valeurs d'un ensemble de types peut être appelé un type abstrait.

# Inférence et vérification de types

Le typage est inféré :

- à la compilation, `OCaML` infère le type de chaque expression
- si le processus d'inférence échoue, alors une erreur de typage est levée.

Le typage est statique : `OCaML` relève les problèmes liés au typage avant l'exécution.

- un certain nombre d'opérateurs sont prédéfinis pour des types prédéfinis (eg `+`, `*.`, ..).
- `OCaml` utilise ces opérateurs pour déterminer le type de certains variables ainsi que le type de l'expression où ils apparaissent
- `OCaML` infère le type le plus général possible (variables de type `'a, 'b, ...`)

# Inférence et vérification de types

```
# let f x = x;;
val f : 'a -> 'a = <fun>
# let f x y =  y;;
val f : 'a -> 'b -> 'b = <fun>
# let f x = x + 1;;
val f : int -> int = <fun>
# let f x = x ^ x;;
val f : string -> string = <fun>
# f 3;;
Error: The constant 3 has type int but an expression
       was expected of type string
# let g x = x ^ (x + 1);;
Error: This expression has type string but an expression
       was expected of type int
```

# Catégories de types

Il existe deux catégories de type :

- les types de *base* (ou atomique)
  `bool, int char, float, string`
- les types *composés* définis à l'aide d'opérateurs de types

Les types (de base ou composés) peuvent être combinés à l'aide d'opérateurs de types pour obtenir de nouveaux types (composés).

Définition récursive de l'ensemble des types !!

# Opérateur de fonction `->`

L'opérateur de type `->` permet de définir de nouveaux types, ceux d'une fonction d'un type vers un autre type

- `int` est un type (de base)
- `int -> int` est un type (composé) à partir de `int` et de `->`
- donc, `(int -> int) -> int` et `int -> (int -> int)` sont deux types composés ... mais différents

```
# let carre x = x * x;;
  val carre : int -> int = <fun>
# sin;;
- : float -> float = <fun>
```

- `carre` fonction prend en paramètre un entier, retourne un entier
- `sin` prend en paramètre un flottant, retourne un flottant

# Parenthésage

Le parenthésage par défaut d'une séquence d'opérateurs `->` est de droite à gauche `a1 -> a2 -> ... an` est équivalent à `a1 -> (a2 -> (... -> (an-1-> an)...)` contrairement à l'appel de fonction qui est de gauche à droite.

```
# max;;
- : 'a -> 'a -> 'a = <fun>
# max 3;;
- : int -> int = <fun>
# max 3 4;;
- : int = 4
```

`max` est une fonction de deux arguments, `max 3` est une fonction d'un argument entier et fera le max entre cet argument et 3. `max 3 4` est un entier.

# Opérateur de produit cartésien `*` (couples)

L'opérateur `*` est l'opérateur de produit (cartésien).

Le produit cartésien de deux ensembles $A$ et $B$ est l'ensemble:

$$A \times B = \{(a, b) \mid a \in A \text{ et } b \in B\}$$

Il permet de représenter l'ensemble des couples d'éléments appartement respectivement à deux types donnés.

`int * int`, `float * int`, `int * char`, `(int -> float) * int`

```
# (2, 3);;
- : int * int = (2, 3)
# 2, 3;;
  - : int * int = (2, 3)
# let t = (3.14,10);;
val t : float * int = (3.14, 10)
# let ascii_a = 97, char_of_int 97;;
val ascii_a : int * char = (97, 'a')
# let c = (fun x -> float_of_int x), 5;;
val c : (int -> float) * int = (<fun>, 5)
```

# Fonctions sur les couples

Utilisation possible des fonctions (prédéfinies) de décomposition `fst` et `snd` de type `a * 'b -> 'a` et `a * 'b -> 'b` respectivement.

```
# fst (2, 3)
- : int 2
# snd (2, 3)
- : int 3

# let couple_square x = x, x*x;;
val couple_square : int -> int * int = <fun>
# let pair x = x, x;;
val pair : 'a -> 'a * 'a = <fun>
```

!! Dans le dernier exemple que le corps de la fonction permet d'inférer que le résultat est un couple mais ne permet pas d'obtenir un type pour `x` plus précis que `'a`.

# Couples et `match ... with`

`(x,y)` peut servir de motif (attention à la linéarité)

```
# let caract_complexe c =
    match c with
     (0.,0.) -> "zero"
   | (0.,x) -> "imaginaire "^string_of_float x
   | (x,0.) -> "réel  "^string_of_float x
   | (x,y) -> (string_of_float x)^"+i"^string_of_float y
val caract_complexe : float * float -> string = <fun>
```

# Décomposition par filtrage de motif

naïvement

```
# let somme_comp t = fst t + snd t;;
val somme_comp : int * int -> int = <fun>
```

avec `let.. in` ou `let` par filtrage (attention à la linéarité)

```
# let somme_comp t = let x,y = t in x + y;;
```

par un couple paramètre (attention à la linéarité)

```
# let somme_comp (x,y)  = x + y;;
```

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

# Produit cartésien généralisé (tuples)

Le produit cartésien binaire se généralise aux tuples.

$$E_1 \times E_2 \times \cdots \times E_n = \{(e_1, e_2, \cdots, e_n) \mid e_i \in E_i,\ 1 \leq i \leq n\}$$

Exemples:

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

Attention : l'opérateur de type `*` n'est pas associatif

```
# 1, (2, 3);;
- : int * (int * int) = (1, (2, 3))
# (1, 2), 3;;
- : (int * int) * int = ((1, 2), 3)
# let x,y,z = 1, (2, 3);;
Error: This expression has type 'a * 'b but an expression
    was expected of type  'c * 'd * 'e
```

# Tuples

Les tuples peuvent aussi être paramètres de fonctions.

```
# let sum3 (i, j, f) = i + j + int_of_float f;;
val sum3 : int * int * float -> int = <fun>
# sum3 (1, 2, 3.);;
- : int = 6
```

Une fonction peut retourner un tuple:

```
# let next (u, v) = v, (u + v);;
val next : int * int -> int * int = <fun>
# next (1,1);;
- : int * int = (1, 2)
# next(next (1,1));;
- : int * int = (2, 3)
# next(next(next (1,1)));;
- : int * int = (3, 5)
```

# Filtrage pour les tuples

On peut récupérer les valeurs d'un tuple par filtrage:

```
# let tuple = 1, "deux", 3.5;;
val tuple : int * string * float = (1, "deux", 3.5)
# let i, str, f = tuple in i + int_of_float f, str;;
- : int * string = (4, "deux")

# let (x,(y,z)) = (1,(2,3));;
val x : int = 1
val y : int = 2
val z : int = 3

# let (x,t) = (1,(2,3));;
val x : int = 1
val t : int * int = (2, 3)
```

# Requête de définition `type`

La requête `type` permet de donner un nom à un type (en général composé)

```
# type complex = float * float;;
type complex = float * float
```

Si nécessaire, il est possible de préciser le type d'un objet : `(expression/variable : type)`

Attention : cela doit être cohérent avec l'inférence de type d'OCaml.

```
# (1., 2.);;
- : float * float = (1., 2.)
# ((1., 2.) : complex);;
- : complex = (1., 2.)
# ((6. +. 1., 2.) : complex);;
- : complex = (7., 2.)
```

# La construction d'un type enregistrement

Au lieu d'utiliser des tuples dans lesquels l'ordre des éléments a une importance, on peut utiliser des enregistrements dans lesquels les éléments sont nommés.

Déclaration d'un type enregistrement

```
type <nom_de_type> =
{
  label1 : type1;
  label2 : type2;
  ...
  labeln : typen;
}
```

On peut ainsi redéfinir le type `complex` vu précédemment:

```
type complex = { real : float; imag : float; }
```

# Constructeurs et accesseurs

Une valeur du type `nom_de_type` s'écrit:

```
{
  label1 = valeur1;
  label2 = valeur2;
  ...
  labeln = valeurn;
}
```

**L'ordre des lignes n'a pas d'importance.**

```
# {real = 1.; imag = 3.14};;
- : complex = {real = 1.; imag = 3.14}
 # let i = {imag = 1.; real = 0.};;
val i : complex = {real = 0.; imag = 1.}
# let c = { imag = 3.; real = 1.0 };;
val c : complex = {real = 1.; imag = 3.}
```

# Accès aux champs d'un enregistrement et filtrage

Étant donnée une valeur de type enregistrement, on *accède* à un des champs en suffixant la valeur par le champs voulu. Exemple:

```
# { real = 1.0; imag = 0. }.real;;
- : float = 1.
# i.imag;;
- : float = 1.
```

On peut filtrer sur les valeurs des champs :

```
# let caract_complexe c =
  match c with
    {real = 0. ; imag = 0.} -> "zero"
  | {real = 0. ; imag = x } -> "imaginaire pur"
  | {real = x ; imag = 0.} -> "reel"
  | {real = x ; imag = y} -> "quelconque" ;;
val caract_complexe : complex -> string = <fun>
```

# Les types somme ou types algébriques

Définition d'un type énuméré de valeurs

```
# type couleur = Rouge | Noir;;
type couleur = Rouge | Noir
# type enseigne = Pique | Trefle | Coeur | Carreau;;
type enseigne = Pique | Trefle | Coeur | Carreau
```

Le séparateur `|` correspond à une union (ou somme) de types.
Les valeurs énumérées doivent débutées par une majuscule

```
# let couleur_enseigne =
    function Pique -> Noir
    | Trefle -> Noir
    | Coeur -> Rouge
    | Carreau -> Rouge;;
val couleur_enseigne : enseigne -> couleur = <fun>
```

# Les types somme ou types algébriques

Certains constructeurs peuvent porter une donnée[4]

```
# type carte = As of enseigne | Valet of enseigne |
                Dame of enseigne | Roi of enseigne |
                Numero of int * enseigne;;
type carte =
    As of enseigne
  | Valet of enseigne
  | Dame of enseigne
  | Roi of enseigne
  | Numero of int * enseigne

# Roi(Coeur);;
- : carte = Roi Coeur
# Numero(4,Trefle);;
- : carte = Numero (4, Trefle)
# As Pique;;
- : carte = As Pique
```

[4] Les valeurs ne sont que des constructeurs sans donnée

# Les types somme ou types algébriques

```
# let valeur carte = match carte with
          Numero(n,_) -> n
        | Valet _ -> 11
        | Dame _ -> 12
        | Roi _ -> 13
        | As _ -> 14;;
val valeur : carte -> int = <fun>

# let bataille c1 c2 = if valeur c1 > valeur c2 then c1
                                                else c2;;
val bataille : carte -> carte -> carte = <fun>
```

# Variable anomyme

Le caractère `_` (souligné ou "tiret du 8" sur un clavier AZERTY) correspond à une variable anonyme.
On peut l'utiliser

- à gauche du `=` dans un `let` ou `let ... in`

```
# let x, _, z = 1,2,3 in x, z;;
- : int * int = (1, 3)
```

- dans un motif d'un `match`

```
let est_une_figure carte =
  match carte with
    As _ -> false
  | Numero(_, _) -> false
  | _ -> true
```

- dans un fichier `.ml` pour évaluer une expression dont le résultat n'a pas d'intérêt

```
let _ = couleur_carte (Numero(7, Pique))
let _ = couleur_carte (As Coeur)
```

# Regroupement des clauses d'un `match`

```
match carte with
  As e -> e
| Roi e -> e
| Dame e -> e
| Valet e -> e
| Numero(_, e) -> e
```

```
match carte with
  As e | Roi e | Dame e | Valet e | Numero(_, e) -> c
```

```
match carte with
  As _ -> false
| Numero(_,_) -> false
| _ -> true
```

```
match carte with
  As _
| Numero(_,_) -> false
| _ -> true
```

# Autre exemple

```
# type form = Square of float | Circle of float
                              | Rectangle of float * float;;
type form = Square of float | Circle of float
                              | Rectangle of float * float
# Rectangle (10., 3.);;
- : form = Rectangle (10., 3.)
# Square(3.4);;
- : formes = Square 3.4
# let perimeter_rect w h = 2. *. (w +. h);;
val perimeter_rect : float -> float -> float = <fun>
# let perimeter_circle r = 2. *. 3.14 *. r;;
val perimeter_circle : float -> float = <fun>
# let perimeter form =
  match form with
    Rectangle(w, h) -> perimeter_rect w h
  | Circle r -> perimeter_circle r
  | Square c -> perimeter_rect c c;;
val perimeter : formes -> float = <fun>
```

# Types récursifs (ou inductifs)

La récursivité est utilisable dans la définition des types. Ceci permet en particulier de construire des types infinis.

On peut par exemple représenter les listes d'entiers[5].

```
# type intlist = NI | CI of int * intlist;;
                (* NI: liste d'entiers vide *)
type intlist = NI | CI of int * intlist

# CI(1, CI(2, CI(3, NI)));;
- : intlist = CI (1, CI (2, CI (3, NI)))
# NI;;
- : intlist = NI
# let est_vide l = match l with
    NI -> true
  | _ -> false;;
val est_vide : intlist -> bool = <fun>
```

[5] Définition peu utile, puisqu'il existe un type prédéfini pour les listes

# Types récursifs (ou inductifs)

Filtrage avec `match ... with`

```
# let head l = match l with
    CI(h,_)  -> h
  | _ -> failwith "liste vide";;
val head : intlist -> int = <fun>
# let tail l = match l with
   CI(_,t)  -> t
  | _ -> failwith "liste vide";;
val tail : intlist -> intlist = <fun>
```

# Types polymorphes

Polymorphisme paramétrique : type paramétré par un autre type

Utilisation de `'a`, `'b`, `'c` pour un type non spécifié ⇒ Généricité

```
# let f x y = x, y+1;;
val f : 'a -> int -> 'a * int = <fun>
```

`'a -> int -> 'a * int` dépend du type paramètre `'a`

Déclaration d'un type paramétré

```
# type 'a couple_gen_int = 'a * int;;
type 'a couple_gen_int = 'a * int
# (("bonjour", 2) : 'a couple_gen_int);;
- : string couple_gen_int = ("bonjour", 2)

# let prem (c : 'a couple_gen_int) = fst c;;
val prem : 'a couple_gen_int -> 'a = <fun>
```

# Types récursifs et polymorphes

Comment définir un type liste générique (liste d'éléments appartenant tous à un même type non fixé)?

```
# type 'a mylist = Nil| CT of 'a * 'a mylist;;
type 'a mylist = Nil | CT of 'a * 'a mylist
# Nil;;
- : 'a mylist = NT
# CT('a', CT('b', CT('c', NT)));;
- : char mylist = CT ('a', CT ('b', CT ('c', NT)))

# CT((1,2), CT ((2,3) , CT((3,4), Nil)));;
- : (int * int) mylist = CT((1, 2),CT((2, 3),CT((3, 4),Nil)
```