Chapitre 2
Les fonctions sont des données comme les autres

Programmation fonctionnelle, Licence d'informatique, 2020

L’inférence de type

- Il n’est pas nécessaire de déclarer les types des valeurs et fonctions, OCaml infère les types les plus généraux

= typage structurel (≠ du typage par nom)

- Exemple :

```
# let rec sub_list the_list =
match the_list with
| [] -> []
| [elem] -> [elem]
| elem1 :: (elem2 :: suite) -> elem1 :: (sub_list suite)
;;
val sub_list : 'a list -> 'a list = <fun>

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

Le polymorphisme

- Une fonction peut s'appliquer sur plusieurs types différents (types 'a ou 'b)
- Exemple :

```
# let swap (val1, val2) =
(val2, val1)
;;
val swap : 'a * 'b -> 'b * 'a = <fun>

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

# swap (2.3, "abc");;
- : string * float = ("abc", 2.3)
```

- polymorphisme ≠ généricité => chaque code instance est compilé séparément
- polymorphisme ≠ héritage => création d'une classe fille pour pouvoir exécuter une méthode mère sur un objet dérivé

Les fonctions en argument

- Une fonction est une donnée comme les autres, on peut la passer en argument d'autres fonctions.

- Exemple :

```
# let pair nb =
(nb mod 2 = 0)
;;
val pair : int -> bool = <fun>

# let rec sub_list (predicat, the_list) =
match the_list with
| [] -> []
| elem :: suite ->
    if (predicat elem)
    then elem :: sub_list(predicat, suite)
    else sub_list(predicat, suite)
;;
val sub_list : ('a -> bool) * 'a list -> 'a list = <fun>

# sub_list (pair, [1; 2; 3; 4]);;
- : int list = [2; 4]
```

Les fonctions comme valeur de retour

- Une fonction peut être la valeur de retour d'autres fonctions.
- Exemple :

```
# let rec sub_list predicat =
let rec sub_list2 the_list =
    match the_list with
    | [] -> []
    | elem :: suite ->
        if (predicat elem)
        then elem :: (sub_list2 suite)
        else (sub_list2 suite)
in sub_list2
;;
val sub_list : ('a -> bool) -> (‘a list -> 'a list) = <fun>

# let pair_sub_list = sub_list pair ;;
val pair_sub_list : int list -> int list = <fun>

# pair_sub_list [1; 2; 3; 4];;
- : int list = [2; 4]
# (sub_list pair) [1; 2; 3; 4];;
- : int list = [2; 4]
```

Les fonctions anonymes

- On peut alléger les notations en utilisant les fonctions anonymes :

function arg -> expr ;; ou **fun** arg1 arg2 … argn -> expr ;;

- Exemple :

```
# let rec sub_list predicat =
fun the_list ->
   match the_list with
   | [] -> []
   | elem :: suite ->
      if (predicat elem)
      then elem :: (sub_list predicat suite)
      else (sub_list predicat suite)
;;
val sub_list : ('a -> bool) -> 'a list -> 'a list = <fun>

# sub_list (fun nb -> nb mod 2 = 1) [1; 2; 3; 4];;
- : int list = [1; 3]
```

Exercice

- Utiliser les fonctions anonymes pour :
    - Écrire une fonction map_list qui prend en argument une fonction et applique cette fonction à tous les éléments d'une liste
    - Utiliser map_list pour écrire une fonction qui incrémente tous les éléments d'une liste d'entiers

Vous prendrez soin de bien préciser les signatures attendues de vos fonctions

Correction

```
# let rec map_list f =
fun the_list ->
match the_list with
| [] -> []
| elem :: suite -> (f elem) :: (map_list f suite)
;;
val map_list : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let incr_list the_list =
map_list (fun nb -> nb + 1) the_list
;;
    val incr_list : int list -> int list = <fun>

# incr_list [1 ; 3 ; 6 ; 7];;
- : int list = [2; 4; 7; 8]
```

Curryfication

- Une fonction classique (décurryfiée) d'arguments, arg_1, arg_2, …, arg_n, est appelée sur les n arguments, qui doivent être fournis simultanément. En fait, elle est appelée sur un argument unique de type n-uplet (arg_1, arg_2, …, arg_n) et le type de la fonction décurryfiée est t_1 * t_2 * … * t_n -> t.

- Curryfiée, elle est d'abord appelée sur arg_1, ce qui produit une fonction qui peut à son tour être appelée sur arg_2, ce qui produit une fonction… qui finalement peut-être appelée sur arg_n et fournit le résultat final. Cette fonction peut donc être partiellement appliquée si on ne lui fourni que ses premiers arguments. Le type de la fonction curryfiée est t_1 -> t_2 -> … -> t_n -> t.

Fonctions curryfiées

- Syntaxe :

let nom_fun arg1 arg2 … argn **=** …

ou **let** nom_fun arg1 arg2 … argn : type1 -> type2 -> … -> typen -> type_res **=** …

- Exemple :

```
# let plus nb1 nb2 =
   nb1 + nb2
;;
val plus : int -> int -> int = <fun>
```

Exercice

- Donnez les types décurryfiés et curryfiés des fonctions suivantes. Précisez à quelles fonctions correspondent leurs applications partielles.

  - La concaténation de deux listes

  - L'ajout de 2 entiers

  - L'insertion, d'un élément dans une liste triée pour un ordre donné (on procèdera à l'application partielle de l'ordre)

  - Découpage d'une liste en 2 sous-listes qui contiennent respectivement les valeurs qui valident ou non un prédicat donné (on procèdera à l'application partielle du prédicat)

Correction

- La concaténation de deux listes
  - ‘a list * ‘a list -> ‘a list
  - (@) : ‘a list -> ‘a list -> ‘a list
  - (@) [1 ; 2 ; 3] : int list -> int list, ajoute la liste [1 ; 2 ; 3] en tête d’une liste d’entiers

- L’ajout de 2 entiers
  - Int * int -> int
  - (+) : int -> int -> int
  - (+) 3 : int -> int, ajoute 3 à un entier

Correction

- L’insertion, d’un élément dans une liste triée pour un ordre donné
- (‘a * ‘a -> bool) * ‘a * ‘a list -> ‘a list
- (‘a -> ‘a -> bool) -> ‘a -> ‘a list -> ‘a list
- insert (<) : ‘a -> ‘a list -> ‘a list, insert un élément dans une liste triée par ordre croissant

Correction

- Découpage d'une liste en 2 sous-listes qui contiennent respectivement les valeurs qui valident ou non un prédicat donné
- (‘a -> bool) * ‘a list -> (‘a list * ‘a list)
- (‘a -> bool) -> ‘a list -> (‘a list * ‘a list)
- split (fun n -> n >= 0) : ‘a list -> (‘a list * ‘a list), découpe une liste d'entiers en 2 listes contenant respectivement les entiers positifs ou nul et strictement négatifs