Programmation Fonctionnelle Avancée

Séance 1 : Polymorphisme, listes, records

Alexandros Singh

Université Paris 8

1er octobre 2023

Que fait la fonction suivante et quel est son type ? :

```
let id x = x;;
```

En l'appliquant à divers exemples, nous obtenons :

```
# id 5;;
- : int = 5
# id 5.0;;
- : float = 5.
# id "Hi!";;
- : string = "Hi!"
# type example_sum_type = Const0 | Const1;;
type example_sum_type = Const0 | Const1
# id Const0;;
- : example_sum_type = Const0
```

C'est un exemple de polymorphisme !

Voyons ce que donne le toplevel :

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

Cela ne ressemble à aucun type que nous avons vu jusqu'à présent. En effet, OCaml essaie de trouver le type le plus général, ce qui implique ici la notion de :

Variables de type

Les fonctions polymorphes ont des signatures impliquant des variables de type qui peuvent, a priori, représenter n'importe quel type.

Pourquoi "a priori" ? Parce que des contraintes peuvent être déduites :

```
# let succ : 'a -> 'a = fun x -> x + 1;;
val succ : int -> int = <fun> (** 'a ne peut être qu'int **)
```

Listes

En regardant dans la bibliothèque standard, nous découvrons que le module List commence par :

```
(* An alias for the type of lists. *)
type 'a t = 'a list = [] | (::) of 'a * 'a list
```

Un type de données récursif et polymorphe ! Une liste est soit la liste vide `[]`, soit constituée d'une paire d'éléments et d'une liste, réalisée à l'aide du constructeur `::`.

```
#[];;
- : 'a list = []
# 1 :: 2 :: 3 :: 4 :: 5 :: [];; (* ou juste [1; 2; 3; 4; 5] *)
- : int list = [1; 2; 3; 4; 5]
```

Récursivité structurelle

La structure récursive des listes suggère une structure récursive pour les fonctions qui les manipulent :

```
# let add_one: int list -> int list = function
    | [] -> []
    | h :: t -> (h+1) :: (add_one t);;
val add_one : int list -> int list = <fun>
# (add_one [1; 2; 3; 4; 5]);;
- : int list = [2; 3; 3; 4; 5]
```

On parle ici de récursivité structurelle : la structure récursive du type de données dicte la structure récursive de l'algorithme.
Nous décrivons le comportement de l'algorithme sur les constructeurs du type `list` et nous laissons la récursivité se chargera du reste.

Nous voulions appliquer une fonction à chacun des éléments d'une liste. Comment faire en utilisant la récursivité structurelle ?

```
# let rec iter f = function (* identique à la version dans List *)
  | [] -> ()
  | a::l -> f a; iter f l;;
val iter : ('a -> 'b) -> 'a list -> unit = <fun>
```

Pourquoi s'agit-il d'une fonction à valeur `unit` ?

```
# (iter (+));;
- : int list -> unit = <fun>
# [1; 2; 3; 4; 5] |> (iter (+));;
- : unit = ()
```

a. `x |> f` est exactement équivalent à `(f x)`.

Utile que pour les effets de bord...

```
# ["Bonjour, "; "Alex!"] |> (iter print_string);;
Bonjour, Alex!
- : unit = ()
```

Nous devrions plutôt collecter les résultats. Naturellement, nous pouvons les rassembler dans une liste :

```
let rec map f l = (*appel récursif pas tail-modulo-cons, mais bon...*)
  match l with
  | [] -> []
  | x :: xs -> (f x) :: map f xs
```

Map d'une fonction anonyme sur une liste :

```
# (map (function x -> x+1) [1; 2; 3; 4; 5]);;
- : int list = [2; 3; 4; 5; 6]
```

Autres fonctions pratiques :

- `List.length : 'a list -> int` : donne la longueur d'une liste
- `List.nth : 'a list -> int -> 'a` : donne le nième élément ($O(n)$)
- `List.append : 'a list -> 'a list -> 'a list` : ajoute deux listes, peut être utilisé en infixe comme `@`. (complexité de `a @ b` : $O($`List.length a`$)$)

Supposons que l'on nous donne une liste de listes et que je veuille l'"aplatir", i.e. concaténer tous ses éléments. Quelle est la signature de la fonction `flatten` dont nous aurons besoin ?

Supposons que l'on nous donne une liste de listes et que nous voulions l'"aplatir", i.e. concaténer tous ses éléments. Quelle est la signature de la fonction `flatten` dont nous aurons besoin ?

```
1 flatten : ('a list) list -> 'a list
```

Comment l'implémenter via la récursivité structurelle ?

- Cas de base :
- Appel récursif :

Supposons que l'on nous donne une liste de listes et que nous voulions l'"aplatir", i.e. concaténer tous ses éléments. Quelle est la signature de la fonction `flatten` dont nous aurons besoin ?

```
1 flatten : ('a list) list -> 'a list
```

Comment l'implémenter via la récursivité structurelle ?

- Cas de base : l'aplatissement d'une liste vide produit une liste vide.
- Appel récursif :

Supposons que l'on nous donne une liste de listes et que nous voulions l'"aplatir", i.e. concaténer tous ses éléments. Quelle est la signature de la fonction `flatten` dont nous aurons besoin ?

```
flatten : ('a list) list -> 'a list
```

Comment l'implémenter via la récursivité structurelle ?

- Cas de base : l'aplatissement d'une liste vide produit une liste vide.
- Appel récursif : l'aplatissement d'une liste non vide consiste à concaténer le premier élément avec le reste de la liste aplatie.

Supposons que l'on nous donne une liste de listes et que nous voulions l'"aplatir", i.e. concaténer tous ses éléments.

```
1 (* identique à List.flatten *)
2 let rec flatten : ('a list) list -> 'a list = function
3     [] -> []
4     | x :: xs -> x @ flatten xs
```

Records

Un enregistrement est composé d'entrées nommées (de n'importe quel type de données) :

```
# type student = {name : string; num : int};;
type student = { name : string; num : int; }
# let alice = {name = "Alice"; num = 1};;
val alice : student = {name = "Alice"; num = 1}
# alice.name;;
- : string = "Alice"
# match alice with {name = n; num = i} -> n;;
- : string = "Alice"
```

Records

Les entrées d'enregistrements dont nous avons parlé jusqu'à présent sont immutables. Une manière pratique de créer de nouveaux enregistrements à partir d'enregistrements existants est :

```
# {alice with num = 5};;
student = {name = "Alice"; num = 5}
# let add_one (s : student)  = match s with
    {name = n; num = i} -> {s with num = i + 1};;
val add_one : student -> student = <fun>
# add_one alice;;
- : student = {name = "Alice"; num = 2}
# alice;;
- : student = {name = "Alice"; num = 1}
```

Records

Les enregistrements comportant des entrées mutables peuvent être déclarés et mis à jour comme suit :

```
# type mutable_student = {name : string; mutable num : int};;
type mutable_student = { name : string; mutable num : int; }
# let mutable_alice : mutable_student = {name = "Alice"; num = 1};;
val mutable_alice : mutable_student = {name = "Alice"; num = 1}
# mutable_alice.num <- 5;;
- : unit = ()
# mutable_alice;;
- : mutable_student = {name = "Alice"; num = 5}
# mutable_alice.name <- "Bob";;
Error: The record field name is not mutable
```

Ecrivez une fonction qui renvoie true si le prenom d'un étudiant est "Alex", sinon false.

Ecrivez une fonction qui renvoie true si le prenom d'un étudiant est "Alex", sinon false.

```
1 let find_alex_0 (s : student) : bool =
2     if s.name = "Alex" then true else false
```

Fonctions sur les records

Ecrivez une fonction, en utilisant le filtrage de motifs, qui renvoie true si le prenom d'un étudiant est "Alex", sinon false.

```
1 let find_alex_1 (s : student) : bool = match s with
2     {name = n; num = _} -> if n = "Alex" then true else false
```

Ecrivez une fonction élégante, en utilisant le filtrage de motifs, qui renvoie true si le prenom d'un étudiant est "Alex", sinon false.

```
1 let find_alex_2 : student -> bool = function
2      {name = "Alex"; _} -> true
3     | _ -> false
```