Informatique *Cours*
Programmation fonctionnelle avec OCaml - Polymorphisme et Ordre supérieur
Ipesup

Objectifs

- Comprendre le polymorphisme et les fonctions d'ordre supérieur.
- Savoir lire et écrire les types polymorphes en OCaml.
- Utiliser les itérateurs principaux du module `List`.

1 Polymorphisme

Le *polymorphisme* désigne la capacité d'un même opérateur ou type à s'appliquer à plusieurs types de valeurs. Par exemple, les opérateurs de comparaison (`=`, `>=`, etc.) s'utilisent sur des expressions de même type, quel qu'il soit. Le type `list` est aussi polymorphe : il peut représenter des listes d'entiers (`int list`), de flottants (`float list`), etc.

Ce principe vaut également pour certaines fonctions comme `mem`, applicable à toute liste dès lors que la valeur recherchée `x` et les éléments de la liste `l` sont du même type. Pour exprimer à la fois ce polymorphisme et la contrainte de type entre `x` et `l`, on introduit des *variables de type*.

Ainsi, le type d'une liste s'écrit α `list`, où α désigne un type quelconque. Par commodité, on note ces variables `'a`, `'b`, etc.

Le type de `mem` est `('a * 'a list) -> bool`, qui prend une valeur et une liste de même type `'a`, et renvoie un booléen.

Les variables de type décrivent donc à la fois des types inconnus et les relations entre arguments et résultat d'une fonction. Par exemple, `fun x -> x` a pour type `'a -> 'a` : elle prend une valeur de tout type et renvoie une valeur du même type.

Elles peuvent aussi apparaître dans la définition de nouveaux types, comme le type prédéfini `'a option`, somme disjointe du constructeur `None` (sans argument) et de `Some` (avec un argument de type `'a`) :

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

Ce type sert à représenter l'absence (`None`) ou la présence (`Some v`) d'une information.

1.1 Inférence et polymorphisme

L'algorithme d'inférence d'OCaml détermine automatiquement le type des expressions polymorphes. Même si son fonctionnement n'est pas au programme MP2I/MPI, il est utile de savoir déduire le type d'une fonction pour interpréter les messages d'erreur du compilateur.

Pour ce faire, on part d'une fonction dont les arguments et le résultat ont des types inconnus, notés par des variables de type. Puis, on affine progressivement ces types en analysant le code.

Exemple

```
let rec f (x, y) =
  match x with
  | [] -> y
  | z :: s -> f (s, z)
```

On suppose d'abord que `f` a le type `'a -> 'b`.
Comme `x` et `y` sont dans une paire, on pose `'a = 'c * 'd`, d'où `f : 'c * 'd -> 'b`.
Le filtrage sur `x` montre que c'est une liste, et que `y` a le même type que la valeur renvoyée ; on obtient alors `'c = 'e list` et `'d = 'b`, soit `f : 'e list * 'b -> 'b`.
Dans la seconde branche, le filtrage `z :: s` indique que `z : 'e` et `s : 'e list`.
L'appel récursif `f (s, z)` impose que `z` ait le même type que `y`, donc `'e = 'b`.
Le type final de `f` est donc le type polymorphe : `'b list * 'b -> 'b`

2 Ordre supérieur

Dans un langage fonctionnel comme OCaml, les fonctions sont des valeurs comme les autres. On peut donc les stocker dans des n-uplets, des enregistrements ou des listes :

Exemple

```
type t = { f : (int -> int) * int; x : char }
let p = ((fun x -> x + 1), 42)
let l = [(fun x -> x + 1); (fun x -> x * x)]
```

Elles peuvent aussi être transmises en argument ou renvoyées comme résultat.

Les fonctions qui prennent d'autres fonctions en entrée ou en sortie sont appelées *fonctions d'ordre supérieur*.

2.1 Fonctions comme arguments

Certaines fonctions reçoivent naturellement d'autres fonctions comme paramètres.

Pour calculer la somme $\sum_{i=1}^{n} f(i)$ pour une fonction quelconque `f`, on peut définir :

```
let rec somme (f, n) =
  if n <= 0 then 0
  else f n + somme (f, n - 1)
```

Le type de `somme` est `(int -> int) * int -> int`.

Exemple

Pour calculer la somme des dix premiers carrés, on appelle :

```
# let v = somme ((fun x -> x * x), 10) ;;
- : int = 385
```

Cet appel évalue $f(1) + f(2) + \cdots + f(10)$, c'est-à-dire $1^2 + 2^2 + \cdots + 10^2 = 385$.

2.2 Fonctions en résultat

Une fonction peut aussi renvoyer une autre fonction.

Pour définir l'application $x \mapsto \frac{f(x+dx)-f(x)}{dx}$ qui approche la dérivée d'une fonction `f` avec un pas `dx`, on écrit :

```
let derive (f, dx) =
  fun x -> (f (x +. dx) -. f x) /. dx
```

Le type inféré est `(float -> float) * float -> (float -> float)`, soit une fonction qui prend une autre fonction et un réel, et renvoie une fonction `float -> float`.

Par associativité à droite de l'opérateur `->`, OCaml affiche :

```
(float -> float) * float -> float -> float
```

2.3 Fonctions à plusieurs arguments

Jusqu'ici, les fonctions à plusieurs arguments étaient définies à l'aide de n-uplets, comme :

```
let plus (x, y) = x + y
```

En exploitant l'ordre supérieur, on peut la réécrire :

```
let plus = fun x -> fun y -> x + y
```

ou, plus simplement,

```
let plus x = fun y -> x + y
```

Ainsi, `plus` prend un entier `x` et renvoie `fun y -> x + y` : son type est `int -> int -> int`.

L'appel `(plus v) w` calcule alors `v + w`, qu'on écrit plus simplement `plus v w` :

```
# let v = plus 32 10
val v : int = 42
```

Cette écriture présente deux avantages.

- Elle évite de créer une paire : l'appel `plus (32,10)` alloue la paire `(32,10)`, la déconstruit, puis libère la mémoire, tandis que `plus 32 10` n'alloue rien.
- Elle permet l'*application partielle*.

Exemple

```
# let f = plus 32
val f : int -> int = <fun>
```

La variable `f` contient alors une fonction équivalente à `fun y -> 32 + y` :

```
# let v1 = f 10
val v1 : int = 42
# let v2 = f 100
val v2 : int = 132
```

L'application partielle permet aussi d'optimiser les calculs répétés.

Exemple

Si une fonction `f (x, y)` effectue un calcul coûteux dépendant seulement de `x`, deux appels `f (e1, e2)` et `f (e1, e3)` répéteraient le même calcul.
La version, d'ordre supérieur :

```
let f x =
  let v = (* Calcul coteux utilisant x *) in
  fun y -> (* expression utilisant x, y et v *)
```

permet de le factoriser :

```
let g = f e1
let v1 = g e2
let v2 = g e3
```

Le calcul de `v` est effectué une seule fois dans `f e1`, puis réutilisé dans les appels de `g`.

Fonctions d'ordre supérieur sur les listes

Le module `List` d'OCaml propose plusieurs *itérateurs*, fonctions d'ordre supérieur permettant d'appliquer une opération à chaque élément d'une liste. Les plus utilisés sont `iter`, `map`, `for_all` et `fold_left`.

iter type `('a -> unit) -> 'a list -> unit`.
Applique une fonction `f` à chaque élément d'une liste.

```
let rec iter f l =
  match l with
  | [] -> ()
  | x::s -> f x; iter f s
```

Exemple

```
# List.iter print_int [1; 2; 3; 4];;
1234
```

map type `('a -> 'b) -> 'a list -> 'b list`.
Construit une nouvelle liste en appliquant `f` à chaque élément.

```
let rec map f l =
  match l with
  | [] -> []
  | x::s -> f x :: map f s
```

Exemple

```
# List.map float_of_int [1; 2; 3; 4];;
- : float list = [1.0; 2.0; 3.0; 4.0]
```

for_all type `('a -> bool) -> 'a list -> bool`.
Teste si un prédicat `p` est vrai pour tous les éléments d'une liste.

```
let rec for_all p l =
  match l with
  | [] -> true
  | x::s -> p x && for_all p s
```

Exemple

```
# for_all (fun x -> x mod 2 = 0) [10; 2; 32; 4];;
- : bool = true
```

fold_left type `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.
Applique `f` de gauche à droite selon `f (... (f (f init e1) e2) ...) en`.

```
let rec fold_left f acc l =
  match l with
  | [] -> acc
  | x::s -> fold_left f (f acc x) s
```

Exemple

```
# let l = [10; 2; 32; 4];;
# List.fold_left (fun x y -> x + y) 0 l;;
- : int = 48
```