Option informatique, cours 3

Structures de données 1/3 : les listes

C. MULLAERT

Lycée Saint-Louis

Année 2024-2025

Plan du cours

Types de données récursifs et/ou polymorphes

Types simples

On rappelle que l'on a vu, dans le premier cours, les types sommes et les enregistrements :

```
type couleurDesYeux = Bleu | Vert | Marron;;
type nouveauType = Entier of int | Paire of int*int ;;
type complexe = {re: float; im: float};;
type complexe = C of float*float;;
```

Ces commandes permettent de définir de nouveaux types à partir des types de base `int` et `float`. Ces types peuvent ensuite être utilisés et manipulés.

On rappelle que les types, comme les champs d'un enregistrement, commencent par une minuscule et que les constructeurs commencent par une majuscule.

Types récursifs

Ocaml permet aussi de définir des types **récursifs**, dans lesquels le type que l'on définit apparait dans la définition. C'est le cas du type liste suivant :

```
type liste = Vide | Liste of int*liste;;
```

Comment donner un sens à cette définition ?

Types récursifs

```
type liste = Vide | Liste of int*liste;;
```

Un élément du type liste peut être :

- soit la constante `Vide`
- soit le couple formé d'un entier et de la constante `Vide`
- soit le couple formé d'un entier avec une liste composée d'un entier et de la constante `Vide`
- et ainsi de suite...

Formellement, on a définit ainsi un type qui contient toutes les suites finies d'entiers : ce sont les **listes** d'entiers.

Types récursifs et polymorphes

La construction suivante des listes d'entiers pourrait se généraliser aux listes de flottants, de booléens ...

```
type liste1 = Vide | Liste of int*liste1;;
type liste2 = Vide | Liste of float*liste2;;
type liste3 = Vide | Liste of bool*liste3;;
```

Ocaml permet de définir tous ces types en une ligne et de leur donner le même nom : c'est le **polymorphisme**. Pour cela, on écrit

```
type 'a liste = Vide | Liste of 'a*'a liste;;
```

Le type `'a` est polymorphe, il peut remplacer n'importe quel autre type. On dit que le type `'a liste` est **paramétré** par le type polymorphe `'a`.

Utilisation des constructeurs

On définit donc le type liste de la façon suivante :

```
type 'a liste = Vide | Liste of 'a*'a liste;;
```

Les éléments de ce type sont donc définis à l'aide des constructeurs `Vide` et `Liste` :

```
let l1 = Vide;; (* liste vide*)
let l2 = Liste (2,Vide);; (* liste 2*)
let l3 = Liste (1,Liste (2,Liste (3,Vide)));;
(* liste 1,2,3*)
```

Longueur d'une liste

Les fonctions qui manipulent des structures de données récursives sont très souvent elles-mêmes récursives, et commencent par un filtrage de la liste à l'aide des constructeur. On peut citer la longueur d'une liste :

```
let rec length = function
  | Vide -> 0
  | Liste(a,l) -> 1 + length l;;
```

Cette fonction est du type `'a liste -> int` et de complexité linéaire en la longueur de la liste.

On dit qu'elle est polymorphe, car elle s'applique indifféremment à toutes les valeurs de type `'a liste`.

Somme des éléments d'une liste

De la même façon, on peut écrire la somme des éléments d'une liste d'entiers :

```
let rec somme = function
  | Vide -> 0
  | Liste(a,l) -> a + somme l;;
```

Cette fonction est du type `int liste -> int` et de complexité linéaire en la longueur de la liste. Elle n'est pas polymorphe.

Réécrire ces deux fonctions sous forme terminale.

Concaténation de listes

On souhaite écrire une fonction qui prend deux listes en argument et qui renvoit la liste concaténée

```
let rec concatene l1 l2 = match l1 with
  | Vide -> l2
  | Liste(a,l) -> Liste(a,concatene l l2);;
```

Cette fonction polymorphe est du type `'a liste -> 'a liste -> 'a liste` et de complexité linéaire en la longueur de la première liste.

Renversement d'une liste

On souhaite écrire une fonction qui prend une liste en argument et qui renvoit une liste du même type, mais où l'ordre des éléments a été inversé.

```
let rec rev = function
  | Vide -> Vide
  | Liste(a,l) -> concatene (rev l) (Liste(a,Vide));;
```

Cette fonction polymorphe est du type `'a liste -> 'a liste`.
Quelle est sa complexité ?

La fonction précédente est de complexité quadratique en la longueur de la liste. Il est possible de réaliser cette opération avec une complexité linéaire :

```
let rev l =
  let rec aux l acc = match l with
        |Vide -> acc
        |Liste(a,q) -> aux q (Liste(a,acc))
  in aux l Vide;;
```

Résumé sur les listes

```
type 'a liste = Vide | Liste of 'a*'a liste;;
```

- Le type liste est un type récursif et polymorphe
- Les fonctions qui manipulent les listes procèdent presque exclusivement par filtrage et appels récursifs
- Les motifs de filtrage reprennent les constructeurs définis dans le type : `Vide` et `Liste` sur notre exemple
- Le plan d'étude général d'une fonction récursive qui manipule les listes est toujours le même : terminaison, correction, complexité
- Ne pas oublier de mentionner la récursivité terminale si c'est le cas

L'implémentation des listes en Ocaml

Le module List

Les types `'a list` (sans e à la fin) est directement disponible, au même titre que les types de base `int`, `float`, ... La définition est très semblable à ce que nous avons vu en première partie du cours :

```
type 'a list = [] | (::) of 'a * 'a list;;
```

Il faut connaître le nom du constructeur pour pouvoir déclarer des liaisons et utiliser le filtrage. Il existe également quelques particularité de syntaxe à connaître.

Le module List

La liste vide s'écrit `[]` et le constructeur est `::`. Il se place en position infixe (à préférer) ou préfixe.

Ainsi, les expressions `(::)(3,[])`, `3::[]` et `1::2::3::[]` ont pour type `int list`.

Pour plus de commodité, ces expressions peuvent être écrites respectivement `[3]` et `[1;2;3]`, mais c'est bien le constructeur `::` qu'il faut utiliser comme motif de filtrage.

Le module List

La longueur d'une liste est déjà implémentée dans la fonction `List.length`. Si on souhaite la redéfinir, on peut écrire

```
let length l=
  let rec aux acc= function
    |[] -> acc
    |a::q -> aux (1+acc) q
  in aux 0 l;;
```

La seule différence avec la fonction length définie plus haut dans le cours est l'utilisation du constructeur `::` et de la liste vide `[]` comme motifs de filtrage.

La fonction de concaténation se réécrit :

```
let rec concatener l1 l2= match l1 with
  |[]->l2
   |h::q -> h :: (concatener q l2);;
```

L'opérateur @ permet de concaténer deux listes. Par exemple `[1;2]@[3]` est équivalent à `1::[2;3]` et à `1::2::[3]`.

Le retournement d'une liste se réécrit :

```
let renverser l =
  let rec aux l1 l2 = match l1 with
        |[] -> l2
        |h::q -> aux q (h::l2)
  in aux l [];;
```

Cette fonction existe déjà sous le nom `List.rev`.

Le module List

Le module List comporte également les fonctions suivantes, qu'il peut être utile de connaitre :

- `List.hd`, de type `'a list -> 'a`, qui renvoit le premier élément (la tête) de la liste et lève une exception si elle est vide
- `List.tl`, de type `'a list -> 'a list`, qui renvoit la liste privée de son premier élément (la queue) et lève une exception si l'argument est vide

Si l'on souhaite omettre le préfixe `List.` pour invoquer les fonctions du module List, il suffit d'exécuter la commande `open List;;`.

L'ensemble des fonctions du module List est consultable à l'adresse `https://ocaml.org/api/List.html`. Ces fonctions ne sont bien sûr pas à connaitre.