Introduction à Caml (3) : Types algébriques et Listes

1 Types Algébriques (Rappels)

1.1 Les exemples

Dans le TP3 nous avons brièvement parlé des types algébriques. Rappelons ici l'exemple du type `option`, très utile :

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

Ainsi que la possibilité de filtrer sur une valeur de type `'a option` :

```
# let result = Some 42;;
result : int option
# match result with
    | None -> "vide"
    | Some i -> "contient le nombre " ^ string_of_int i;;
- : string = "contient le nombre 42"
```

1.2 Sémantique des Types Algébriques

Une fois un type algébrique défini, on peut caractériser l'ensemble de ses valeurs (c'est-à-dire l'ensemble des valeurs Caml qui ont ce type). Chaque *constructeur* du type algébrique prend 0 ou plus arguments, et renvoie une valeur qui est différente de toute valeur venant d'un autre constructeur.
Par exemple, si on represente des monnaies (incompatibles entre elles par défaut) :

```
type monnaie =
    | Euro of float
    | Dollar of float;;

let x = Euro 17. ;;
let y = Dollar 17. ;;
x = y;;  (* renvoie false ! *)
```

On voit que `x` et `y` sont des valeurs de type `monnaie`, mais ne sont pas égaux. Le type algébrique `monnaie` est l'union disjointe des valeurs `Euro f` et des valeurs `Dollar f` pour `f` n'importe quelle valeur parmi les flottants.
Ceci permet de ne pas mélanger des sommes d'argent exprimées dans différentes monnaies, et donc de ne pas se tromper. On peut imaginer la même chose pour les dimensions physiques.
Si on souhaite additionner des dollars et des euros, on peut toutefois écrire une fonction de conversion vers une monnaie, et effectuer l'addition dans cette unique monnaie (par exemple l'euro), de cette manière :

```
let convertir_en_euro x = match x with
    | Euro y -> y
    | Dollar z -> z *. 0.72 ;;   (* au cours actuel *)

let ajouter x y =
    Euro (convertir_en_euro x +. convertir_en_euro y);;
```

Ici on voit que la fonction `ajouter` est de type `monnaie -> monnaie -> monnaie`, et renvoie toujours une quantité exprimée en euros.

2 Raisonner sur les Types Algébriques

2.1 Rappel : relation d'ordre strict

On dit qu'une relation binaire R est

Irréflexive si $\forall x,\ \neg R(x,x)$

Transitive si $\forall x\ y\ z,\ R(x,y) \wedge R(y,z) \Rightarrow R(x,z)$

Bien fondée s'il n'existe pas de suite infinie $(x_i)_{i\in\mathbb{N}}$ telle que $\forall i \in \mathbb{N},\ R(x_i, x_{i+1})$.

Si une relation R est à la fois irréflexive et transitive on parle d'une *relation d'ordre strict*. On peut définir un tel ordre strict et bien fondé sur tout type algébrique en décidant que les constructeurs constants (sans arguments, teles que `None` ou `[]`) sont minimaux et que pour un constructeur A *of* $t_1 * \cdots * t_n$ on a $\forall x_i : t_i, A(x_1, \ldots, x_n) \succ x_i$. Par exemple, sur le type liste, la relation $\succ$ est un ordre strict bien fondé dont l'élément minimal est `[]` :

— $\forall x\ l,\ x :: l \succ l$

— $\forall l_1\ l_2\ l_3,\ l_1 \succ l_2 \wedge l_2 \succ l_3 \Rightarrow l_1 \succ l_3$.

On notera cet ordre $\succ_{liste}$ par la suite.

2.2 Induction Structurelle

Tout comme la récurrence mathématique habituelle sur les entiers naturels, les types algébriques se prêtent à ce qu'on appelle un *raisonnement par induction*, et plus précisément *induction structurelle*.
Informellement, l'induction structurelle permet de prouver qu'une propriété P est valide sur tous les éléments d'un type algébrique τ si la propriété est valide

pour les constructeurs constants de τ (`None`, `[]`, etc.), et si elle se transmet par les constructeurs non constants [1].

2.3 Entiers de Peano

Remarque : les entiers naturels peuvent être représentés (naïvement) par un type algébrique — on parle généralement des *entiers de Peano*. Ce type comprend un constructeur constant, représentant 0, et un constructeur de successeur. Tout nombre $n \in \mathbb{N}$ peut être représenté par $s^n(0)$ où s est le symbole du successeur (décomposition en base 1).

```
type nat =
    | Zero
    | Succ of nat;;
```

Dans ce cas, l'induction structurelle correspond exactement à la récurrence mathématique : $\forall P, (P(0) \wedge (\forall n. P(n) \Rightarrow P(n+1))) \Rightarrow \forall n, P(n)$.
Les opérations habituelles peuvent s'écrire par récurrence, par exemple la somme :

```
# let rec somme x y = match x with
    | Zero -> y
    | Succ x' -> Succ (somme x' y);;
somme : nat -> nat -> nat
```

3 Listes

On a vu que Caml fournissait un type de *listes*, dont les constructeurs (spéciaux) sont la liste vide `[]` et le constructeur infixe `::`. La notation `[1;2;3;4]` est en fait un raccourci pour `1 :: (2 :: (3 :: (4 :: [])))`. Le schéma d'induction structurelle sur les listes est :
$\forall P, (P(\texttt{[]}) \wedge (\forall xl, P(l) \Rightarrow P(x :: l))) \Rightarrow \forall l, P(l)$.

3.1 Fonctions basiques

3.1.1 Longueur

Voyons quelques exemples de fonctions sur les listes. Tout d'abord, pour calculer la longueur d'une liste :

```
let rec list_length = function
    | [] -> 0
    | _::l' -> 1 + list_length l';;
```

Cette fonction termine car l'appel récursif se fait sur la queue de la liste, qui est plus petite dans $\succ_{liste}$. Elle est prouvée correcte par une induction triviale.

1. L'induction structurelle est en fait un cas particulier d'induction dite *bien fondée* qui s'appuie sur un ordre bien fondé ; en effet on a vu qu'à un type algébrique on pouvait associer un tel ordre.

3.1.2 Map

On peut aussi définir un combinateur appelé `map`. Il sert à appliquer une fonction à chaque élément d'une liste et à retourner la liste des résultats. Son type est donc `('a -> 'b) -> 'a list -> 'b list`.

```
let rec list_map f = function
    | [] -> []
    | x :: l' ->
        let y = f x in
        y :: list_map f l';;
```

3.1.3 Concaténation

Concaténer deux listes se fait via la fonction `append`, de type `'a list -> 'a list -> 'a list`, ainsi définie :

```
let rec append l1 l2 = match l1 with
    | [] -> l2
    | x :: l1' -> x :: (append l1' l2);;
```

Elle existe déjà dans Caml, sous la forme de l'opérateur `@` : évaluer `[1;2] @ [3;4]` renvoie bien `[1;2;3;4]`.

3.1.4 Renversement

Renverser une liste se fait avec `rev` (notez la fonction auxiliaire) :

```
let rec rev_append acc l = match l with
    | [] -> acc
    | x :: l' -> rev_append (x :: acc) l';;

let list_rev l = rev_append [] l;;
```

La fonction `rev_append` termine car son argument `l` décroit strictement à chaque appel récursif, dans l'ordre bien fondé $\succ_{liste}$. Prouvons maintenant la correction de cette fonction. Pour cela nous spécifions mathématiquement le résultat attendu, par la définition d'une fonction $\mathfrak{R}$ telle que

$$\mathfrak{R}(l) = \begin{cases} \mathfrak{R}(l')@[x] & \text{si } l = x :: l \\ [] & \text{si } l = [] \end{cases}$$

Prouvons maintenant la propriété $P(l_2) \stackrel{\text{def}}{\equiv} \forall l_1,\ \texttt{rev\_append}\ l_1\ l_2 = \mathfrak{R}(l_2)@l_1$. Par induction sur l_2, commençons par le cas de base $l_2 = []$, qui est trivial (`rev_append` renvoie `acc`, c'est-à-dire l_1 et donc $l_1@[]$). Le cas inductif se prouve ainsi : supposons la propriété vraie pour l_2 (pour tout l_1) et montrons que pour tout x, elle est aussi vraie pour $x :: l_2$. Par la définition de `rev_append` on a `rev_append` $l_1\ (x :: l_2) =$ `rev_append` $(x :: l_1)\ l_2$, en appliquant l'hypothèse d'induction on a `rev_append` $(x :: l_1)\ l_2 = \mathfrak{R}(l_2)@(x :: l_1)$ et donc

`rev_append` l_1 $(x :: l_2) = \mathfrak{R}(l_2)@(x :: l_1) = \mathfrak{R}(x :: l_2)@l_1$ par définition de $\mathfrak{R}$. L'induction est terminée. Maintenant, puisque `rev l = rev_append [] l`, on voit que `list_rev` l = `rev_append []` $l = \mathfrak{R}(l)@$`[]` $= \mathfrak{R}(l)$ et donc `list_rev` est correcte.

3.1.5 Fold

Plus généralement, un combinateur extrêmement utile est appelé `fold` ou `fold_left`. Il sert à parcourir une liste en utilisant une fonction et un « accumulateur »servant de résultat intermédiaire, et peut être utilisé pour définir de nombreux autres combinateurs. C'est un peu l'équivalent fonctionnel d'une boucle « for ».

```
let rec fold_left f acc l =
    match l with
    | [] -> acc
    | x :: l' ->
        let acc' = f acc x in
        fold_left f acc' l';;
```

Le type de cette fonction est `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`. Intuitivement, `fold_left f x [l1;l2;...;ln]` correspond à l'application imbriquée `f (... (f (f x l1) l2) ...) ln`. Elle est suffisamment puissante pour permettre d'écrire plusieurs fonctions vues jusqu'ici :

```
let list_length l =
    fold_left (fun n _ -> n+1) 0 l;;

let rev_append l1 l2 =
    fold_left (fun acc x -> x :: acc) l1 l2;;

let list_rev l =
    fold_left (fun acc x -> x :: acc) [] l;;
```

3.1.6 For all

On peut tester si tous les éléments d'une liste vérifient un certain prédicat, en utilisant la fonction `list_forall : ('a -> bool) -> 'a list -> bool`. On pourra écrire son dual, `list_exists` (du même type) en exercice.

```
let rec list_forall p = function
    | [] -> true
    | x :: l' -> p x && list_forall p l';;
```

3.2 Listes d'association

On peut utiliser les listes pour associer des *clés* à des *valeurs*, comme dans un annuaire où des numéros (ou adresses) seraient associés au nom de leur

propriétaire. Une *liste d'association* est une valeur de type `('a * 'b) list` avec `'a` le type des clés et `'b` le type des valeurs.
On peut définir plusieurs fonctions utiles. Commençons par la fonction `associer` qui, étant donné une liste `l` et une clé `x`, renvoie l'élément associé à `x`, ou `None`.

```
# let rec associer x = function
    | [] -> None
    | (y, value) :: _ when x = y -> Some value
    | _ :: l' -> associer x l';;
val associer : 'a -> ('a * 'b) list -> 'b option
```

et la fonction `enlever`, qui enlève la clé `x` de la liste (si elle est présente) :

```
# let rec enlever x = function
    | [] -> []
    | (y::value) :: l' ->
        if x = y
            then enlever x l'
            else (y::value) :: enlever x l';;
val enlever : 'a -> ('a * 'b) list -> ('a * 'b) list
```