Polynômes

Samuel Mimram

4 février 2006

Ce TD constitue une introduction à la programmation impérative. On se propose principalement de réaliser une implémentation du calcul des opérations usuelles sur les polynômes.

1 Version impérative (tableaux)

1.1 Tableaux et références

1.1.1 Tableaux

Un tableau est un ensemble fini de valeurs du même type indexées par des entiers (de façon injective et tel que l'ensemble des indexes soit un segment initial de $\mathbb{N}$). On crée un tableau avec la fonction `Array.make` (de type `int -> 'a -> 'a array`). Cette fonction prend deux arguments :

1. le nombre d'éléments du tableau ;
2. la valeur initiale des éléments du tableau.

On peut aussi déclarer explicitement un tableau comme dans l'exemple suivant

```
# let a = [|3; 5; 6; 2|];;
val a : int array = [|3; 5; 6; 2|]
```

La valeur `a` déclarée est un tableau de quatre entiers (3, 5, 6 et 2). On accède au n-ième élément du tableau de la façon suivante :

```
# a.(2);;
- : int = 6
```

(la valeur de l'élément d'indice 2 est 6). Attention les indices commencent à 0.

On peut obtenir la longueur d'un tableau grâce à la fonction `Array.length` :

```
# Array.length a;;
- : int = 4
```

L'indice du dernier élément du tableau `a` est donc `(Array.length a) - 1` (notez bien le `- 1` dû au fait que les indices commencent à 0).

► Que ce passe-t-il si vous essayez d'accéder à l'élément d'indice 25 du tableau `a` ?

On peut modifier la valeur d'un élément d'un tableau à l'aide de la façon suivante :

```
# a.(1) <- 12;;
- : unit = ()
# a;;
- : int array = [|3; 12; 6; 2|]
```

(la valeur de l'élément d'indice 1 est maintenant 12).

► Essayez de manipuler un peu les tableaux.

1.1.2 Références

Il est possible d'avoir en caml des valeurs modifiables : les références. La commande suivante déclare `x` comme une référence qui vaut initialement 0 :

```
# let x = ref 0;;
val x : int ref = {contents = 0}
```

On accède au contenu d'une référence grâce à l'opérateur `!` :

```
# !x;;
- : int = 0
```

On peut modifier le contenu d'une référence grâce à l'opérateur `:=` :

```
# x := 5;;
- : unit = ()
# !x;;
- : int = 5
```

`x` contient maintenant la valeur 5.

▶ Essayez de manipuler un peu les références.

Voici par exemple une fonction qui permet de sommer les éléments d'un tableau :

```
let sum_arr a =
  let n = ref 0 in
    for i = 0 to (Array.length a) - 1
    do
      n := !n + a.(i)
    done;
    !n
```

N'hésitez pas à me poser des questions si tout ne vous parait pas clair dans cette fonction.

1.2 Polynômes représentés par des tableaux

1.2.1 Exponentiation rapide

▶ Programmez une fonction récursive `pow_lin` de type `int -> int -> int` telle que `pow_lin a b` vaut a^b. Combien d'étapes de calcul votre fonction nécessite-t-elle ?

▶ Programmez une fonction `pair` de type `int -> bool` qui vaut `true` si et seulement si son argument est pair (on pourra s'aider de l'opérateur infixe `mod` qui calcule un modulo).

▶ Programmez une fonction `pow` de type `int -> int -> int` telle que `pow a b` soit le résultat de l'exponentiation rapide de `a` par `b`. En quoi est-elle plus performante que `pow_lin` ?

1.2.2 Évaluation

On représente un polynôme par le tableau de ses indices (par ordre de puissance croissant). Ainsi, $5x^3 + 10x^2 - x + 9$ sera reprsenté par le tableau `[|9 ; -1 ; 0 ; 10 ; 5|]`. On considèrera que les polynômes ne sont définis que sur les entiers (pas besoin d'utiliser les opérations sur les flottants donc).

▶ En vous inspirant de l'exemple qui calcule la somme des éléments d'un tableau programmez une fonction `eval` de type `int array -> int -> int` qui donne le résultat de l'évaluation d'un polynôme en un point. Vous utiliserez `pow` pour calculer les puissances.

► En réalité, on peut implémenter cette fonction plus efficacement qu'en utilisant l'exponentiation rapide car on doit a priori calculer toutes les valeurs des x^i. On peut donc mieux faire en maintenant la valeur courante de x^i dans une référence `xi` et en la mettant à jour à chaque itération de la boucle en utilisant `xi := !xi * x`.

Implémentez cette version. Comprenez vous bien en quoi elle est de meilleure complexité que la précédente ?

Indice : $\sum_{i=1}^{n} \log(i) \approx O(n \cdot \log(n))$, sauriez-vous le démontrer ?

Quelle aurait été la complexité de la fonction si vous aviez utilisez l'implémentation naïve de l'exponentiation ?

► (méthode de Hörner) En fait on peut encore mieux faire en constatant que

$$p_0 x^0 + p_1 x^1 + \ldots + p_n x^n$$

peut aussi s'écrire

$$(((\ldots((p_n)x + p_{n-1})x \ldots)x + p_1)x + p_0)$$

Implémentez une nouvelle version de la fonction en utilisant cette écriture. Quelle est la nouvelle complexité de votre fonction ?

1.2.3 Affichage

► Programmez une fonction `show` de type `int array -> string` qui renvoie une chaîne de caractères correspondant à l'affichage d'un polynôme par exemple :

```
# show [|1; 2; 3|];;
- : string = "1 * x^0 + 2 * x^1 + 3 * x^2"
```

On rappelle que la concaténation de chaîne de caractères se fait à l'aide de l'opérateur infixe `^` (accent circonflexe) et que l'on peut obtenir la représentation d'un entier sous forme de chaîne de caractères à l'aide de la fonction `string_of_int` de type `int -> string`.

1.2.4 Multiplication par un scalaire, dérivée, somme

► Implémentez la multiplication d'un polynôme par un scalaire `scal` de type `int array -> int -> int array`. La multiplication ne doit pas se faire en place (pourquoi ?), il faut donc penser à allouer un nouveau tableau pour le résultat.

► Implémentez `deriv : int array -> int array` qui calcule la dérivée d'un polynôme.

► Implémentez la fonction `add` de type `int array -> int array -> int array` qui calcule la somme de deux polynômes en supposant que ces deux polynômes ont la même taille.

► Modifiez la fonction précédente pour qu'elle gère correctement l'addition de deux polynôme de degré différent.

1.2.5 Multiplication

► Petit rappel : si $P(X) = \sum_{k=0}^{n} p_k X^k$ et $Q(X) = \sum_{k=0}^{m} q_k X^k$ alors $(P \times Q)(X) = \sum_{k=0}^{n+m} r_k X^k$ avec $r_k = \sum_{i+j=k} p_i \cdot q_j$ (avec $0 \leq i \leq n$ et $0 \leq j \leq m$). Utilisez cette définition pour implémenter la multiplication de deux polynômes `mult` de type `int array -> int array -> int array`. Quelle est la complexité de votre fonction ?

2 Version fonctionnelle (avec des listes)

À partir de maintenant nous allons représenter les polynômes par des `int list`. Le polynôme $\sum_{k=0}^{n} p_k X^k$ sera représenté par la liste `[pk ; ... ; p1 ; p0]` (notez bien que l'ordre a « changé »).

▶ Voici un extrait de la documentation caml :

```
val map : ('a -> 'b) -> 'a list -> 'b list

List.map f [a1; ...; an] applies function f to a1, ..., an , and builds the list
[f a1; ...; f an] with the results returned by f.
```

Reprogrammez une fonction `map` correspondant à cette spécification.

▶ En utilisant la fonction `map`, implémentez la multiplication d'un polynôme par un scalaire `scal` de type `int list -> int -> int list`.

▶ Et hop voici un autre extrait de la doc caml :

```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn
```

Réimplémentez une fonction `fold_left`.

▶ En utilisant cette fonction et en « passant » la valeur courante de x^i. Implémentez l'évaluation d'un polynôme en un point.

▶ Vous commencez à avoir l'habitude :

```
val rev : 'a list -> 'a list

List reversal.
```

Réimplémentez cette fonction.

▶ En utilisant l'égalité $(\sum_{k=0}^{n} p_k X^k)(Q(X)) = p_0 \cdot Q(X) + X \cdot (\sum_{k=1}^{n} p_{k-1} X^{k-1})$, programmez la multiplication de deux polynômes.

3 S'il vous reste du temps

N'hésitez pas à me demander des précisions sur les questions, elles sont volontairement évasives.

▶ Programmez un tri-fusion en utilisant des tableaux.

▶ Programmez un quicksort.