TD 2 - supplément : Rattrapage Caml

Algorithmes, types de données et preuves, Année 2012/2013

Le but de ces exercices est de revoir les fondements de l'écriture de fonctions en Caml et de faire connaissance avec quelques astuces pour rendre l'exécution plus efficace.

1 Fonctions numériques

1.1 Factorielle

1. Écrivez la fonction `fact` qui prend un seul argument (un nombre naturel n) et calcule $n!$. Par convention, $0! = 1$.
2. Étant donnée la définition :

   ```
   let rec fact_acc (n, acc) =
     if n = 0
     then acc
     else fact_acc (n - 1, n * acc)
   ;;
   ```

   donnez l'expression qui permet de calculer $n!$. Autrement dit, avec quelle valeur `v` faut-il appeler `fact_acc` pour que `fact_acc(n, v)` soit égal à `fact(n)` ?

   On appelle la définition de `fact_acc` une définition avec *accumulateur*, parce que les résultats partiels sont accumulés dans la variable `acc`.
3. Vous pouvez tracer l'exécution des deux fonctions avec

   ```
   #trace fact
   ```

   respectivement

   ```
   #trace fact_acc
   ```

   Ici, # fait partie de la commande et n'est pas l'invite de Caml.

1.2 Fibonacci

1. Définissez la fonction `fib` de Fibonacci, où `fib(0) = 0`, `fib(1) = 1` et pour tout `n`, on a `fib(n + 2) = fib(n) + fib(n + 1)` (bien sûr, il faut l'écrire différemment en Caml).
2. Mesurez le temps d'exécution pour un n assez "élevé", par exemple $n = 20$, $n = 30$, $n = 40$:

   ```
   let t1 = Sys.time () in
   let _ = fib 20 in
   let t2 = Sys.time () in
   "temps d'execution: "^string_of_float(t2 -. t1)
   ;;
   ```

3. Suivant le principe de la programmation avec accumulateur du § 1.1, définissez la fonction `fib_acc (n, acc0, acc1)` qui mémorize dans `acc0` et `acc1` les deux derniers résultats calculés jusqu'à maintenant. Avec quels valeurs `v0, v1` faut-il "initialiser" les paramètres `acc0, acc1` pour que `fib_acc (n, v0, v1)` soit égal à `fib(n)` ? Mesurez encore une fois le temps d'exécution.

2 Fonctions sur des listes

2.1 Inversion d'une liste

1. Comme préparatif, programmez la fonction `append` qui concatène deux listes et qui fonctionne comme suit :

```
# append ([1; 2], [3; 4; 5]) ;;
- : int list = [1; 2; 3; 4; 5]
```

2. Écrivez la fonction `rev` qui inverse une liste. Cette fonction fait une récursion simple sur la liste et utilise la fonction `append`.

```
# rev [1; 2; 3; 4; 5] ;;
- : int list = [5; 4; 3; 2; 1]
```

3. Écrivez la fonction `rev_acc` qui inverse une liste, et qui prend deux arguments : la list à inverser, et un accumulateur. Cette fonction n'utilise pas la fonction `append` (ou son homologue prédéfini `@`). Comment faut-il initialiser l'accumulateur pour avoir le même comportement que pour `rev` ?
4. Pourquoi cette fonction est-elle plus efficace que la fonction `rev` ?

2.2 Liste doublement chaînée

Dans des langages impératifs avec pointeurs tels que C ou Java, on peut relativement facilement programmer des listes doublement chaînées : Chaque élément de cette liste (sauf le premier et le dernier) a un pointeur vers son prédecesseur et son successeur. On peut ainsi facilement insérer ou supprimer des éléments près d'un pivot situé à l'intérieur de la liste, sans parcourir la liste toute entière.

Le fragment fonctionnel pur de Caml ne connaît pas de pointeurs, mais on peut simuler une liste doublement chaînée avec un pivot, en scindant la liste doublement chaînée en deux sous-listes, dont la première est inversée.

Par exemple, la liste `[-5; -3; -1; 2; 4; 5]` avec le pivot devant `2` est représentée par le couple de listes `([-1; -3; -5], [2; 4; 5])`. Si on déplace le pivot devant `5`, on obtient `([4; 2; -1; -3; -5], [5])`.

Dans la suite, nous nous intéressons uniquement à des listes traditionnelles triées en ordre croissant. Si nous parlons de liste doublement chaînée (dc), nous nous référons à la structure de données justement évoquée. Tout élément de la

liste gauche est strictement inférieur au pivot, tout élément de la liste droite est supérieur égal au pivot.

1. Écrivez la fonction `dc_to_list` qui convertit une une liste doublement chaînée en liste traditionnelle. Vous pouvez utiliser les fonctions `append` et `rev` ou `rev_acc`.

   ```
   # dc_to_list ([-1; -3; -5], [2; 4; 5]) ;;
   - : int list = [-5; -3; -1; 2; 4; 5]
   ```

2. Écrivez la fonction `move_pivot` qui prend une liste dc et un pivot et produit la liste dc dont le pivot a été déplacé.

   ```
   # move_pivot (([-1; -3; -5], [2; 4; 5]), 5) ;;
   - : int list * int list = ([4; 2; -1; -3; -5], [5])
   # move_pivot (([-1; -3; -5], [2; 4; 5]), -2) ;;
   - : int list * int list = ([-3; -5], [-1; 2; 4; 5])
   ```

3. Utilisez `move_pivot` pour écrire la fonction `list_to_dc` qui prend une liste traditionnelle et un pivot et produit la liste dc correspondante.

   ```
   # list_to_dc [-5; -3; -1; 2; 4; 5] 2 ;;
   - : int list * int list = ([-1; -3; -5], [2; 4; 5])
   # list_to_dc [-5; -3; -1; 2; 4; 5] 7 ;;
   - : int list * int list = ([5; 4; 2; -1; -3; -5], [])
   ```

4. Écrivez la fonction `insert_at_pivot` qui prend une liste dc et un élément et l'insert dans la sous-liste droite.

5. Écrivez la fonction `insert_sorted` qui insère un élément "au bon endroit" dans une liste dc et en fait le nouveau pivot. Utilisez `move_pivot` et `insert_at_pivot`.

   ```
   # insert_sorted (([-1; -3; -5], [2; 4; 5]), 3) ;;
   - : int list * int list = ([2; -1; -3; -5], [3; 4; 5])
   ```

6. Écrivez la fonction `sort_dc` qui prend une liste dc et une liste traditionnelle et insère tous les éléments de la liste tradionnelle dans la liste dc, en appelant `insert_sorted`.

   ```
   # sort_dc ([-1; -3; -5], [2; 4; 5]) [8; 9; 11; -2; -3; -4; 6; 7] ;;
   - : int list * int list =
   ([6; 5; 4; 2; -1; -2; -3; -3; -4; -5], [7; 8; 9; 11])
   ```

   Argumentez pourquoi cette fonction est plus efficace qu'une fonction de tri traditionnelle si elle doit traiter une liste contenant des "blocs" d'éléments déjà triées, comme c'est le cas dans l'exemple.