Université Paris Diderot
M1 Informatique

Programmation Fonctionnelle Avancée
Année 2015-2016

TP n°1

Structures de données fonctionnelles efficaces

Listes à accès arbitraire rapide

On s'intéresse ici à des structures de données purement fonctionnelles (on dit également *persistantes*) permettant de représenter des *listes*, c'est-à-dire des séquences finies et ordonnées d'éléments ayant des opérations “par la gauche” efficaces: ajout via `cons` et retrait via `head` et `tail`. En fait, dans ce qui suit, au lieu de fonctions séparées `head` et `tail`, on utilisera désormais une unique fonction `decons` telle que `decons l = (head l, tail l)`.

Tout en gardant les bonnes propriétés des fonctions précédentes, on souhaite en plus disposer de fonction `nth` et `set_nth` qui soient de complexité logarithmique[1].

Exercice 1 On appelle *b-liste* une liste (usuelle) d'arbres binaires parfaits ayant les données aux feuilles, satisfaisant de plus l'invariant suivant: la taille des arbres binaires est strictement croissante quand on avance vers la droite de cette liste.

1. Donner un type OCaml correspondant à cette structure, ainsi que quelques exemples de petites listes de ce type. Peut-on avoir deux b-listes différentes contenant les mêmes données ?

2. Implémentez les opérations `cons`, `decons`, `nth` et `set_nth` de telle façon que toutes ces opérations soient logarithmiques. On prendra soin d'ajuster la structure de données pour éviter tout calcul superflu de taille d'arbres binaires.

3. (Facultatif) Coder une opération `drop` de complexité logarithmique telle que `drop`(k,l) retourne la b-liste l privée de ses k premiers arguments.

Cette première structure montre donc qu'il est possible d'accéder en temps logarithmique à une position arbitraire dans nos listes. Malheureusement ce premier essai conduit à une perte de complexité des opérations “à gauche” (`cons` et `decons`), qui ne sont plus en temps constant. Nous allons maintenant voir comment remédier à cela. Mais tout d'abord, un petit détour par de l'arithmétique : notre première structure était très liée à la décomposition binaire des nombres, et pour faire mieux nous allons utiliser une autre décomposition moins usuelle.

Exercice 2 On appelle *écriture quasi-binaire* (ou qb-écriture) d'un nombre entier sa décomposition comme somme de nombres de la forme $2^k - 1$ où $k > 0$. On demande de plus que tous les termes de la somme soient différents, à part éventuellement les deux plus petits.

1. Ecrire une fonction `decomp` qui donne l'écriture quasi-binaire de tout nombre entier n. Peut-on avoir plusieurs écritures quasi-binaires du même nombre ?

[1] On parle ici de complexité en temps, dans le pire cas, exprimée en fonction de la taille de la liste.

2. Ecrire deux fonction `next` et `pred` qui font passer de l'écriture quasi-binaire d'un nombre à celle de son successeur et de son prédecesseur, sans chercher à reconstituer le nombre en question, et le tout en temps constant.

On admet que la qb-écriture canonique de n fait intervenir au plus $\lceil lg(n+1) \rceil$ termes, où lg est le logarithme en base 2.

Exercice 3

1. En vous inspirant de ce qui précède, proposer une structure de qb-liste pour laquelle `cons` et `decons` ont des complexités constantes tandis que `nth` et `set_nth` sont logarithmiques.
2. Comparer ces qb-listes avec les listes usuelles : quelles raisons peuvent expliquer la faible adoption des qb-listes en lieu et place des listes usuelles ?

Références:

- Okasaki, chapitre 9 du livre “Purely Functional Data Structures”.
- Okasaki, article de 1995 : “Purely Functional Random-Access Lists”.