TP 4 de programmation fonctionnelle en Objective Caml

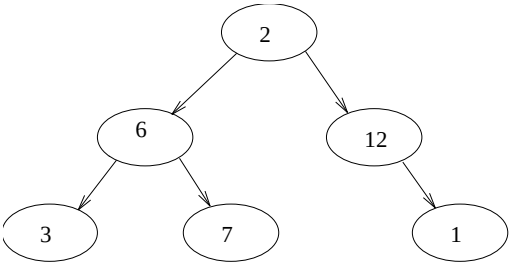
Christian Rinderknecht

3 février 2014

L'objectif est de présenter les arbres binaires et les arbres binaires de recherche. La programmation de ces deux structures de données et les opérations associées sont un bon exercice pour découvrir la puissance d'Objective Caml par rapport à des langages non fonctionels où il faut manipuler moult pointeurs. En particulier, on approfondira l'usage du filtrage et des types polymorphes récursifs, déjà entrevus lors du travail dirigé concernant les listes.

1 Arbres binaires

Un arbre binaire est un arbre où chaque nœud a au plus deux fils, un fils gauche et un fils droit. De plus, chaque nœud est annoté par une étiquette, qui est une valeur a priori quelconque. Voici un exemple d'arbre annoté par des étiquettes de type entier :



Le type des arbres binaires est défini par :

```
type 'a arbre =
  Vide
| Noeud of 'a * ('a arbre) * ('a arbre)
```

Ce type est *paramétré* par la variable de type `'a`, ce qui permet de définir le type des arbres indépendamment du type des étiquettes. On pourra ensuite utiliser le type `int arbre` lorsque l'on veut annoter chaque nœud par un entier, etc.

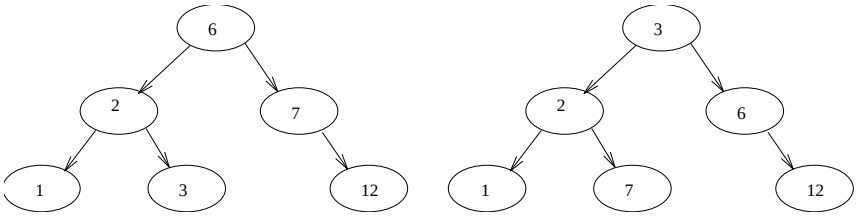
— Écrire une fonction qui calcule la hauteur d'un arbre.
— Écrire une fonction qui calcule la taille (le nombre de nœuds) d'un arbre.
— Écrire une fonction `recherche`, qui prend en argument une étiquette `x` et un arbre `a`, et indique si cette étiquette apparaît sur l'un des nœuds de `a`.

— Écrire une fonction qui énumère les étiquettes d'un arbre et les place dans une liste. On écrira trois versions de cette fonction : en ordre préfixe, infixe, et suffixe. Dans l'ordre préfixe, on commence par placer l'étiquette dans la liste, puis le sous-arbre gauche et enfin le sous-arbre droit. Dans l'ordre infixe, on place l'étiquette entre les sous-arbres, et dans l'ordre postfixe, après les sous-arbres. Par exemple, le parcours de l'arbre ci-dessus en ordre infixe donne la liste `[ 3; 6; 7; 2; 12; 1 ]`.
— Écrire une fonction `compare_arbres` qui compare deux arbres. En plus des arbres à comparer, elle prendra en argument une fonction de comparaison `r` sur les étiquettes. La fonction devra renvoyer `true` si, et seulement si, les deux arbres ont la même structure et si les étiquettes sont deux à deux dans la relation `r`.
— Application : écrire une fonction qui compare deux arbres en ne prenant en compte que leur structure (i.e. sans s'occuper des étiquettes).
— Écrire une fonction `sous_arbre` qui prend deux arbres en argument, et indique si le premier est un sous-arbre (éventuellement égal) de l'autre.

2 Arbres binaires de recherche

Nous allons voir maintenant quelques applications des arbres : les arbres de recherche, qui permettent de classer des données plus efficacement que dans une liste.

Un arbre de recherche est un arbre binaire tel que :
— toutes les étiquettes du sous-arbre de gauche (resp. de droite) sont inférieures (resp. supérieures) à l'étiquette de la racine,
— récursivement, les deux sous-arbres sont eux-mêmes des arbres de recherche.



Le premier arbre ci-dessus est un arbre de recherche. Le second n'en est pas un car 7 est plus grand que 3 mais il se trouve dans le sous-arbre de gauche de la racine.

Dans une liste, les opérations classiques (par exemple, déterminer si un élément appartient ou non à la liste) demandent un temps $O(n)$, c'est-à-dire proportionnel à la taille de la liste, parce qu'on risque d'avoir à parcourir toute la liste pour effectuer la recherche. Dans un arbre binaire de recherche, les éléments sont triés, ce qui permet des opérations plus rapides. En effet, le temps maximal nécessaire à une recherche sera la hauteur de l'arbre, soit $O(\log n)$ si l'arbre est équilibré.

— Écrire une fonction qui renvoie le plus petit élément d'un arbre binaire de recherche. On lancera l'exception `Not_found` si l'arbre est vide.

— Écrire une fonction qui indique si une étiquette donnée apparaît dans un arbre de recherche donné. On tirera parti du fait que l'arbre est classé pour que la recherche soit plus rapide que dans un arbre binaire quelconque.
— Écrire une fonction qui ajoute une étiquette dans un arbre de recherche de façon à ce que celui-ci reste un arbre de recherche. Cela se fait en rajoutant une feuille à l'arbre.
— On dispose déjà de plusieurs fonctions capables de transformer le contenu d'un arbre binaire en liste, en le parcourant dans l'ordre préfixe, infixe ou postfixe. Laquelle de ces fonctions faut-il utiliser pour que la liste résultante soit triée par ordre croissant ?
On en déduit alors un algorithme de tri de liste. On part d'un arbre de recherche vide dans lequel on insère successivement toutes les valeurs de la liste à trier. Il ne reste plus qu'à afficher par ordre croissant les étiquettes de cet arbre grâce à la fonction précédente. L'ordre dans lequel on réalise les insertions est-il significatif ?
— Écrire une fonction qui recherche une étiquette donnée dans un arbre, et la retire. Si l'étiquette n'existe pas, on lancera l'exception `Not_found`. Si elle existe, on renverra donc un nouvel arbre de recherche qui ne contient plus cette étiquette. Indication : on commencera par écrire une fonction qui prend en argument un arbre et retourne la paire constituée de l'élément minimal (s'il n'existe pas alors `Not_found`) et de l'arbre initial sans cet élément.