Arbres

I Différents types d'arbres.

I.1 Arbres binaires généraux

Définition 1

Soit `'a` un type. On appelle ensemble des arbres binaires étiquettés par `'a` (dans la suite on parlera plus simplement de `'a arbre`) le plus petit ensemble tel que :

- l'arbre `Vide` est un `'a arbre` ;
- pour tout `x:'a`, pour tous `fg,fd:'a arbre` l'arbre `Noeud(x,fg,fd)` est un `'a arbre`.

Terminologie : on rappelle quelques définitions vues en TD.

- Lorsqu'un arbre est de la forme `Noeud(x,fg,fd)` où `x : 'a` et `fg, fd : 'a arbre_binaire`, alors `x` s'appelle l'**étiquette** de l'arbre, `fg` s'appelle le **fils gauche** de l'arbre et `fd` s'appelle le **fils droit** de l'arbre.
- Pour a un arbre, on appelle **sous-arbre de *a***, ou **nœud de *a*** un arbre non vide obtenu à partir de a en effectuant un nombre arbitraire de fois les opérations `fils_gauche` et `fils_droit` (sans obtenir d'exception). Le nœud a lui-même (obtenu en effectuant zéro fois les opérations) est appelé la **racine** de a.
- Un nœud de a dont les deux fils sont vides est appelé **une feuille**. Un **nœud interne** d'un arbre a est un nœud de a qui n'est pas une feuille.
- On appelle **profondeur d'un nœud** le nombre de fois où on a dû appliquer l'une des opérations `fils_gauche` ou `fils_gauche` à partir de l'arbre initial pour obtenir ce nœud.

Implémentation immuable : il s'agit d'un ensemble défini inductivement, son implémentation naturelle en Caml se fait par la création d'un type inductif.

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

Exemples 1 : Des arbres binaires peuvent intervenir naturellement en informatique, en mathématiques, ou ailleurs (même si souvent d'autres types d'arbres interviennent aussi). Par exemple :

Implémentation modifiable : on peut aussi implémenter un élément de type `'a arbre` par un tableau `t` d'éléments de type `'a option`, de longueur suffisamment grande (au moins 2^{h+1} avec h la hauteur de l'arbre, pour anticiper sur la section suivante). On utilise l'astuce suivante : l'étiquette de la racine (si elle n'est pas vide) est stockée en `t.(0)` ; et, pour chaque nœud dont l'étiquette est stockée en `t.(i)`, l'étiquette de son fils gauche (s'il est non vide) sera stocké en `t.(2i+1)` et celle de son fils droit (s'il est non vide) en `t.(2i+2)` ; les sous-arbres `Vide` étant représentés par `None`.

Exemple 2 :

L'intérêt de la représentation modifiable est bien sûr qu'on peut modifier en place un arbre déjà défini ; mais son gros défaut est d'imposer une hauteur maximale qu'on ne peut en revanche pas modifier.

I.2 Arbres binaires stricts non vides

Définition 2 : (vue en TD)

On appelle arbre binaire strict un arbre binaire dont tous les nœuds ont 0 ou 2 fils non vides, c'est-à-dire dont tous les nœuds internes ont leurs deux fils non vides.

Exemples 3 :

Dans un arbre binaire strict, aucun nœud n'a de fils vide, sauf les feuilles. En décidant d'écarter l'arbre vide (ce n'est pas une très grosse perte), on a donc deux types d'arbres binaires stricts : les feuilles, et les nœuds internes (sans enfants vides). Oui, c'est une nouvelle définition inductive ! Défaut (peu grave) : on a perdu l'arbre vide. Avantage : on n'a plus besoin d'étiquetter les feuilles et les nœuds par le même type.

Implémentation immuable : cela conduit à l'implémentation suivante.

```
type ('a,'b) abs = Feuille of 'a | Noeud_interne of 'b * ('a,'b) abs * ('a,'b) abs
```

Exemples 4 :

Implémentation modifiable : on peut garder l'idée vue pour les arbres binaires quelconques. Quitte à choisir un élément de type `'a` ou `'b` pour initialiser le tableau, on n'a même plus besoin du type `'a option`. Mais, à moins de se limiter au cas des `('a,'a) abs`, on aura besoin d'un type `('a,'b) etiquette` :

```
type ('a,'b) etiquette = F of 'a | N of 'b
```

I.3 Arbres généraux non vides

Enfin, on peut accepter que certains nœuds d'un arbre aient strictement plus de deux fils. Cela conduit à la définition suivante :

Définition 3

Soit `'a` un type. On appelle ensemble des arbres généraux non vides étiquettés par `'a` (dans la suite : `'a ag`) le plus petit ensemble tel que :

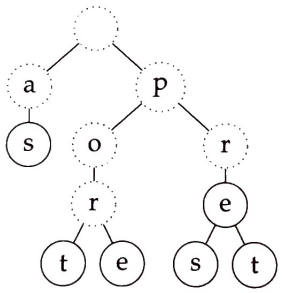
— pour tout `x:'a`, pour tout `lf:'a ag list` l'arbre `Noeud(x,lf)` est un `'a ag`.

Implémentation immuable :

```
type 'a ag = Node of 'a * 'a ag list
```

Exemple 5 :

Exemple 6 : En modifiant légèrement la définition des arbres généraux, on peut obtenir une structure de donnée permettant de stocker de façon compacte un ensemble de mots. Explicitement, une *trie* (ou un *arbre préfixe*) est un arbre dans lequel les arcs d'un nœud vers ses fils sont étiquettés par des caractères. Certains nœuds sont marqués : pour ceux-ci, en lisant la suite de lettres conduisant de la racine à ce nœud (d'où le "préfixe"), on doit obtenir l'un des mots que l'on souhaite stocker. Plus de *tries* en TP.



À gauche : une *trie* permettant de stocker l'ensemble de mots {as, port, pore, pré, près, prêt} (sans les accents).

II Taille et hauteur

II.1 Définitions

Proposition-Définition 4

On définit récursivement la hauteur d'un arbre comme le successeur du maximum des hauteurs de ses fils.
Il faut également un cas d'arrêt qui peut, suivant le type d'arbres, porter sur l'arbre vide (de hauteur -1) ou sur les feuilles (de hauteur 0).

Pour tout arbre non vide, la hauteur est la profondeur maximale d'un nœud de l'arbre.

Démonstration. Par induction structurelle (cf TD).

Il faut savoir coder une fonction hauteur :

— sur les arbres binaires (cf TD) ;

— sur les arbres binaires stricts non vides (c'est très facile d'adapter le TD) ;

— sur les arbres généraux (on le fait tout de suite) :

Définition 5

Pour a un arbre, on appelle taille de $\boldsymbol{a}$, et on note $\boldsymbol{|a|}$, le nombre de nœuds de a.

Même chose : il faut savoir coder une fonction hauteur sur les arbres binaires (cf TD), mais aussi sur les arbres binaires stricts non vides et sur les arbres généraux.

II.2 Propriétés

Proposition 1

Soit a un arbre de taille $|a|$ et de hauteur $h(a)$.

1. Si a est un arbre binaire, on a : $h(a) < |a| < 2^{h(a)+1}$, et donc $\lg(|a|+1) < h(a) < |a|$.
2. Si a est un arbre binaire strict, on a : $2h(a) < n < 2^{h(a)+1}$, et donc $\lg(|a|+1) < h(a) < \frac{|a|}{2}$.
3. Pour un arbre général, on ne peut rien dire...

Démonstration. Par induction structurelle (cf TD).

Représentons les cas extrémaux :

Définition 6

Avec une variante pour les arbres binaires non stricts, on peut ainsi définir les arbres binaires stricts filiformes, peignes à gauche, peigne à droite, parfaits et quasi-complets :

Remarque 1 : On montre aussi, par induction structurelle toujours, qu'un arbre binaire strict non vide a exactement une feuille de plus qu'il n'a de nœuds internes.

III Parcours d'arbres

Comme le montre leur représentation usuelle, on peut voir les arbres comme des cas particuliers de graphes. On peut donc effectuer sur les arbres des parcours en profondeur ou en largeur comme sur les graphes.

Mais c'est beaucoup plus simple pour les arbres parce que .....

III.1 Parcours en profondeur infixe, préfixe, suffixe

Rappel : le principe du parcours en profondeur est de parcourir récursivement chaque fils avant de passer au suivant. Quant à la racine, on peut l'examiner **avant** ses fils : on a alors un parcours en profondeur préfixe. On peut aussi examiner la racine **après** ses fils : on a alors un parcours en profondeur préfixe. Dans le cas d'un arbre binaire, on peut aussi parcourir récursivement le fils gauche, puis examiner la racine, puis parcourir récursivement le fils droit : on a alors un parcours en profondeur infixe.

Exemple 7 : Prenons l'exemple de l'arbre d'une expression arithmétique. Que retrouve-t-on ?

Exercice 1. Implémenter naïvement le parcours en profondeur préfixe d'un `abs` par récursivité enveloppée. Quel est le problème ?

III.2 Parcours en profondeur : l'arbre cache la forêt

Pour implémenter **efficacement** un parcours en profondeur, on va bien sûr se ramener à de la récursivité terminale. Pour ce faire, l'idée, simple, est la suivante : *l'arbre cache la forêt*. On écrit une fonction qui parcours en profondeur non pas un arbre, mais une forêt, c'est-à-dire une liste d'arbres. Ainsi :

- on considère une liste d'arbres qui initialement a un seul élément, l'arbre que l'on souhaite parcourir, et une liste d'éléments initialement vide, correspondant à les étiquettes des nœuds explorés pendant le parcours ;
- lorsqu'on rencontre un arbre non vide dans la liste, on ajoute l'étiquette de son nœud au début de la liste des éléments, puis on rajoute au début de la liste d'arbres les fils gauche et droit de l'arbre ;
- lorsque la liste d'arbres est vide, on renverse et renvoie la liste d'éléments.

Je vous laisse me confirmer que c'est la même idée qu'en ITC.

Exercice 2. Implémenter à l'aide de cet algorithme le parcours en profondeur préfixe d'un `abs`.

III.3 Parcours en largeur : il nous faudrait... des files !

Pour le parcours en largeur, on serait bien embêté pour le coder par récursivité enveloppée ! Mais la solution en passant par la forêt s'adapte par contre sans problème, il suffit de rajouter les fils gauche et droits **à la fin** de la liste d'arbres plutôt qu'au début.

Exercice 3. Le faire et vérifier. Mais... quel est le problème ?

Comme vous l'avez vu en ITC, on résout ce problème en ne manipulant pas des **listes** d'arbres, mais des **files** d'arbres, les files étant une structure de données pour laquelle l'insertion d'un élément en **fin** de file est une opération peu coûteuse. Vous avez vu que cette structure existe en Python. Et en Caml ?

Ça s'appelle un cliffhanger.