Le petit guide du bouturage, ou comment réaliser des arbres mutables en OCaml

Jean-Christophe Filliâtre

CNRS
LRI Université Paris Sud 91405 Orsay, France
Inria Saclay–Île-de-France
jean-christophe.filliatre@lri.fr

Résumé

Un langage de programmation fonctionnelle s'avère un outil de choix pour manipuler des arbres de façon purement applicative. D'une part, les arbres y sont représentés de manière naturelle par un type algébrique et manipulés de façon concise et élégante par le filtrage. D'autre part, le typage statique assure la bonne formation des arbres et exclut tout risque d'avoir omis un cas de figure dans une définition de fonction.

Si on cherche en revanche à manipuler des arbres *mutables*, c'est-à-dire qui peuvent être modifiés en place, tout en restant dans le cadre d'un langage fonctionnel, il semble qu'on doive renoncer partiellement à ces atouts. C'est d'autant plus regrettable que l'immense majorité de la littérature algorithmique ne considère que des arbres mutables. Dans cet article, nous explorons différentes représentations d'arbres mutables dans le contexte du langage OCaml. Nous les comparons en termes de performance, mais aussi en termes d'élégance et de sûreté.

1. Introduction

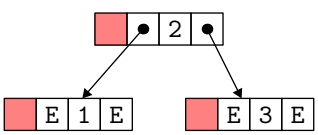
Un langage de programmation fonctionnelle est un outil de choix pour manipuler des structures algébriques telles que des arbres. Pour cette raison, il est particulièrement agréable de l'utiliser comme support pour enseigner l'algorithmique. Dans le langage OCaml, par exemple, une structure d'arbre binaire contenant des entiers en ses nœuds est définie en quelques mots :

```
type tree = E | N of tree * int * tree
```

Ils se lisent comme « une valeur de type `tree` est soit la constante `E` (représentant ici l'arbre vide), soit le constructeur `N` (représentant ici un nœud) appliqué à trois arguments, à savoir un arbre, un entier et un autre arbre ». Ainsi on peut construire un arbre contenant trois nœuds avec l'expression suivante :

```
N (N (E, 1, E), 2, N (E, 3, E))
```

En mémoire, chaque application du constructeur `N` donne lieu à l'allocation d'un bloc de quatre mots, le premier étant réservé par le système et les trois autres étant utilisés pour stocker les trois arguments du constructeur. L'expression ci-dessus donne lieu à la construction de la valeur suivante :



Ce n'est pas différent de ce que nous aurions construit à l'aide de pointeurs, de structures ou encore de classes dans des langages tels que `C`, `C++` ou encore Java. Mais il y a cependant une différence de taille : dans le langage OCaml, un tel arbre est *immuable*. Cela signifie qu'une fois construit, il ne peut être modifié. Si on veut insérer un nouvel élément dans cet arbre, on doit reconstruire un nouvel arbre. Bien entendu, on peut partager de nombreux sous-arbres entre l'ancien et le nouveau. C'est tout l'intérêt des structures immuables que de pouvoir être partagées, sans risque d'aliasing. Pour enseigner l'algorithmique, le caractère immuable de l'arbre est en fait un atout : il n'y a pas à se poser de question telle que « ma fonction doit-elle modifier l'arbre ou au contraire en renvoyer un nouveau » et il n'y a pas non plus de risque de mutation accidentelle.

L'intérêt de la programmation fonctionnelle va plus loin encore. Le typage statique effectué par le compilateur garantit la bonne formation d'un arbre. Plus précisément, il garantit que toute valeur de type `tree` est soit `E`, soit de la forme `N`(l, x, r) avec l de type `tree`, x de type `int` et r de type `tree`. Par ailleurs, un langage comme OCaml fournit une construction de *filtrage* qui permet de procéder par examen sur une valeur de type `tree` de façon extrêmement concise et efficace. Ainsi on peut écrire une fonction qui supprime le plus petit élément dans un arbre binaire de recherche, supposé non vide, aussi simplement que

```
let rec remove_min = function
  | E           -> invalid_arg "arbre vide"
  | N (E, _, r) -> r
  | N (l, x, r) -> N (remove_min l, x, r)
```

Une telle définition se lit comme une définition par cas : si l'argument de `remove_min` est de la forme `E` alors exécuter le code `invalid_arg "arbre vide"` (ici pour lever une exception) ; si en revanche, il est de la forme `N (E, _, r)`, c'est-à-dire un nœud avec un sous-arbre gauche vide (`E`), un élément quelconque à la racine (`_` dénote une valeur quelconque) et un sous-arbre droit que l'on appelle `r`, alors renvoyer `r` ; si enfin l'arbre est un nœud de la forme `N(l,x,r)`, alors renvoyer un nouvel arbre obtenu en évaluant l'expression `N (remove_min l, x, r)`. L'intérêt du filtrage va au delà de sa concision syntaxique. Le compilateur vérifie que tous les cas de figure sont bien couverts et non redondants. Si par exemple, on omet le premier cas, le compilateur le signalera. Si par exemple on remplace `E` par une variable `l` dans le second cas, alors le compilateur signalera que le troisième cas ne sera jamais examiné. Là encore, c'est un outil de choix pour le programmeur qui souhaite manipuler des structures algébriques comme les arbres, et en particulier celui qui enseigne/apprend l'algorithmique.

Cependant, on est forcé de constater que, pour l'immense majorité, la littérature algorithmique ne considère que des structures mutables[1]. Les ouvrages de référence, comme ceux de Knuth [3], Sedgewick [5] ou Cormen *et al* [2], ignorent tout simplement la programmation à base de structures immuables — seul Knuth mentionne brièvement les structures persistantes à la toute fin du volume 3 de *The Art of Computer Programming*. Les ouvrages qui traitent effectivement de l'algorithmique dans le contexte d'un langage fonctionnel sont quasi inexistants ; le livre de Chris Okasaki, *Purely Functional Data Structures* [4], fait exception.

Pour le programmeur OCaml qui souhaiterait mettre en œuvre les concepts d'un ouvrage traditionnel d'algorithmique se pose alors une question intéressante : comment réaliser des structures mutables, pour ne pas trahir les algorithmes présentés, sans pour autant renoncer aux bénéfices du typage statique et du filtrage ? La question est plus subtile qu'il n'y paraît. Cet article sans prétention a pour but de donner quelques éléments de réponse.

Pour prendre un exemple très concret, considérons les arbres binaires de recherche tels qu'ils sont présentés en Java dans l'excellent *Algorithms, 4th edition* de Sedgewick et Wayne [5]. Un extrait est donné figure 1 page 4. Nous l'avons légèrement simplifié ici pour ne considérer que des arbres contenant des entiers et changer la structure de dictionnaire en simple structure d'ensemble. La classe

1. L'adjectif « mutable » est du français correct qui signifie bien « qui peut être muté » (et pas seulement en parlant d'un fonctionnaire).

`Tree` définit un arbre binaire, la valeur `null` étant utilisée pour représenter l'arbre vide. La méthode statique `add` insère un élément `x` dans un arbre `t`. L'insertion se fait en place. Néanmoins, la méthode `add` renvoie une valeur de type `Tree`, significative lorsque `t` vaut `null` (dans le cas contraire, c'est la valeur de `t` inchangée). On note l'écriture récursive de `add`, qui modifie les pointeurs `left` et `right` en remontant. Un certain nombre d'autres opérations sur les arbres binaires de recherche sont écrites de la même façon (elles ne sont pas données ici). Puis une valeur de type `Tree` est encapsulée dans une autre classe, `BST`, afin de fournir les opérations sur les arbres binaires de recherche sous forme de méthodes (dynamiques).

Nous nous proposons d'écrire un code équivalent dans le langage OCaml.

2. Arbres mutables

Dans cette section, nous présentons quatre façons différentes de réaliser des arbres mutables dans le langage OCaml. Pour chaque solution, l'objectif est d'écrire un module réalisant une structure d'ensemble à l'aide d'un arbre binaire de recherche mutable. Le module doit avoir signature suivante :

```
module type SET = sig
  type t
  val create: unit -> t
  val add: int -> t -> unit
  val mem: int -> t -> bool
  val remove: int -> t -> unit
  val size: t -> int
end
```

Le type `t` cache donc un arbre mutable. Vu qu'on s'intéresse ici à une structure d'ensemble, il n'y aura pas de doublons dans nos arbres. Le code OCaml présenté dans cet article est accessible à l'adresse `https://www.lri.fr/~filliatr/pub/mtrees.ml`.

2.1. Avec un pointeur nul

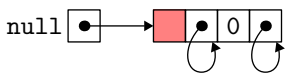
Une première solution consiste à reproduire en OCaml un code identique au code Java donné dans l'introduction (page 4). On commence donc par introduire un type `node` pour représenter un nœud de l'arbre.

```
  type node = { v: int; mutable left: node; mutable right: node; }
```

Le champ `v` contient la valeur stockée en chaque nœud. Les champs `left` et `right` sont déclarés comme mutables ; c'est là l'objectif. On définit ensuite une valeur `null` pour représenter l'arbre vide.

```
  let rec null = { v = 0; left = null; right = null; }
```

On exploite ici la possibilité offerte par OCaml de construire une valeur récursive. Cette valeur est construite une fois pour toute. Schématiquement, on vient de construire ceci :



où la boîte de gauche représente la variable de programme `null` (une variable globale, typiquement) et la boîte de droite la valeur de type `node` allouée sur le tas. Les champs `left` et `right` de ce bloc pointe vers ce même bloc, de manière cyclique ; c'est le sens du `let rec` utilisé dans la définition

```
// arbres binaires de recherche
// null représente l'arbre vide
class Tree {
  int elt;
  Tree left, right;

  Tree(Tree left, int elt, Tree right) {
    this.left  = left;
    this.elt   = elt;
    this.right = right;
  }

  static Tree add(Tree t, int x) {
    if (t == null) return new Tree(null, x, null);
    if      (x < t.elt) t.left  = add(t.left,  x);
    else if (x > t.elt) t.right = add(t.right, x);
    return t;
  }
  ...
}

// l'arbre est ensuite encapsulé dans une classe
class BST {
  private Tree root;
  BST() { this.root = null; }
  void add(int x) { this.root = Tree.add(this.root, x); }
  ...
}
```

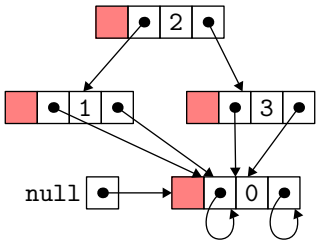
FIGURE 1 – Arbres binaires de recherche en Java, à la manière d'*Algorithms* [5].

de `null`. On notera que, dans nos dessins, chaque flèche désigne un bloc sans faire particulièrement attention à l'endroit précis où la tête de flèche est dessinée. Ce n'est pas important ici.

Pour construire un arbre contenant trois nœuds, on peut, par exemple, écrire le code suivant :

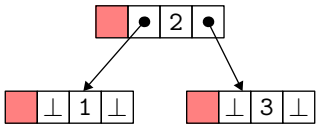
```
let t1 = { left = null; v = 1; right = null; }
let t3 = { left = null; v = 3; right = null; }
let t2 = { left = t1;   v = 2; right = t3;   }
```

Trois blocs sont alloués en mémoire. Les quatre occurrences de la valeur `null` sont autant de pointeurs identiques, ce que l'on peut visualiser ainsi :



De manière générale, un arbre contenant N nœuds occupe $4N$ mots en mémoire, sous la forme de N blocs sur le tas, de 4 mots chacun. (On ne décompte pas la valeur `null`, allouée une fois pour toute.)

On peut souhaiter simplifier les schémas en utilisant une notation particulière pour la valeur du pointeur `null`. Si on choisit la notation usuelle $\perp$, alors on pourra se représenter cet arbre plus simplement comme



Cependant, il faut bien garder à l'esprit la présence du bloc représentant `null` et, surtout, son caractère cyclique. Si on tente par exemple d'afficher la valeur de l'arbre `t2` dans le *toplevel* OCaml, on va obtenir une valeur infinie, sans distinction entre le nœud représentant `null` et les autres. On est seulement sauvé par la profondeur maximale de l'affichage des valeurs. Pour la même raison, si on tente une comparaison structurelle comme `t2=t2`, alors on obtient une erreur :

```
Out of memory during evaluation.
```

De manière générale, on ne peut pas utiliser d'égalité structurelle sur les valeurs de type `node`. Il faut se limiter à l'égalité physique. En particulier, on *doit* utiliser l'égalité physique pour tester si une valeur de type `node` est ou non la valeur `null`, exactement comme on le fait dans le code Java. Ainsi, l'insertion d'un élément `x` dans un arbre `t` s'écrit de façon quasi identique au code Java :

```
let rec add x t =
  if t == null then { v = x; left = null; right = null; } else begin
    if      x < t.v then t.left  <- add x t.left
    else if x > t.v then t.right <- add x t.right;
    t
  end
```

Un léger avantage de la valeur `null` ici est d'être typée de type `node`. Ainsi on ne peut la comparer qu'à des valeurs de ce type-là.

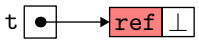
De même que le code Java de la page 4 encapsule l'objet de type `Tree` dans une classe `BST`, on peut ici aussi encapsuler la valeur de type `node` dans une référence et réaliser ainsi un module de signature `SET` en quelques lignes.

```
module S : SET = struct
  type t = node ref
  let create () = ref null
  let add x s = s := add x !s
  ...
```

Si on construit un arbre vide avec la déclaration suivante

```
let t = S.create ()
```

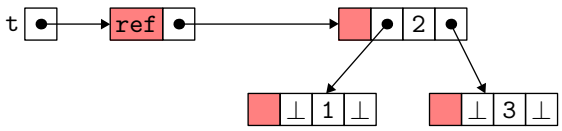
on a construit ceci en mémoire :



Si ensuite on ajoute successivement les entiers 2, 1 et 3 à cet arbre `t` avec

```
let () = S.add 2 t; S.add 1 t; S.add 3 t
```

alors on parvient à la situation suivante :



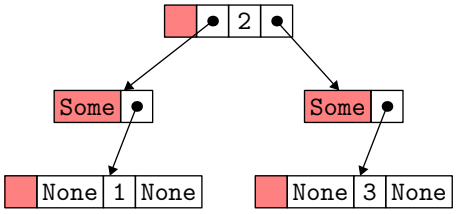
Cette solution est relativement simple à mettre en œuvre, car elle coïncide mot à mot avec le code impératif traditionnel. Mais elle en a également tous les défauts. En particulier, c'est la responsabilité du programmeur de traiter correctement le cas du pointeur `null`. Le compilateur OCaml n'est ici d'aucun secours.

2.2. Avec un type `option`

Pour remédier au défaut que nous venons juste d'évoquer, une autre solution consiste à utiliser des types `option` pour les deux champs `left` et `right` du type `node`, c'est-à-dire

```
type node = { v: int; mutable left: node option; mutable right: node option; }
```

La valeur `None`[2] représente le pointeur nul, c'est-à-dire l'arbre vide, et la valeur `Some n` un sous-arbre non vide dont la racine est le nœud `n`. Ainsi un arbre contenant trois nœuds aura la forme suivante :



2. On rappelle que le type `option` d'OCaml, fourni par la bibliothèque `Pervasives`, est défini par `type 'a option = None | Some of 'a`.

Comme rappelé sur cette figure, la valeur `None` est représentée de manière immédiate par un mot (un entier, en fait) alors qu'une valeur de la forme `Some n` est représentée par un bloc de taille 1 sur le tas. L'occupation mémoire totale est donc de $6N-2$ mots pour un arbre contenant N nœuds : chaque nœud contribue pour 4 mots et les $N-1$ nœuds qui ne sont pas la racine contribuent chacun pour 2 mots supplémentaires stockant les valeurs `Some` correspondantes.

Si cette représentation est plus gourmande que la précédente, elle a l'indéniable avantage de distinguer clairement l'arbre vide. On retrouve en particulier tous les bénéfices du typage statique d'OCaml. Ainsi la fonction d'insertion d'un élément `x` dans un arbre se voit obligée d'examiner le cas de l'arbre vide explicitement :

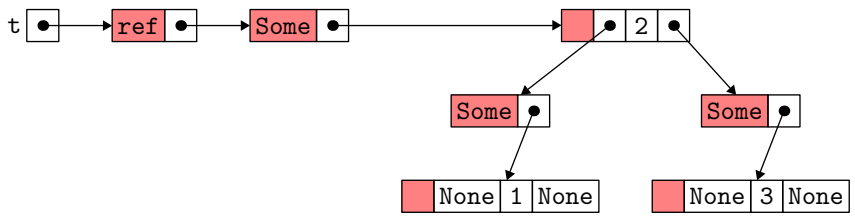
```
let rec add x = function
  | None ->
      Some { v = x; left = None; right = None }
  | Some t as o ->
      if      x < t.v then t.left  <- add x t.left
      else if x > t.v then t.right <- add x t.right;
      o
```

Le reste du code n'est pas différent. On note que le constructeur `Some` est alloué seulement dans le premier cas, où une feuille est construite. Dans le cas d'une insertion récursive, le constructeur `Some` est réutilisé ; c'est le rôle du `as o` dans le second filtrage.

Comme précédemment, on encapsule un tel arbre dans un module, pour lui donner la signature `SET` attendue. Là encore, une référence est nécessaire. On écrit donc

```
module S : SET = struct
  type t = node option ref
  let create () = ref None
  let add x s = s := add x !s
  ...
```

Ainsi un arbre `t` construit de la même façon que dans la section précédente amène à la situation suivante :



La présence du constructeur `Some` à la racine porte donc l'occupation mémoire à $6N$ mots pour un arbre contenant N nœuds. (On omet la référence, comme précédemment.)

2.3. Avec une référence

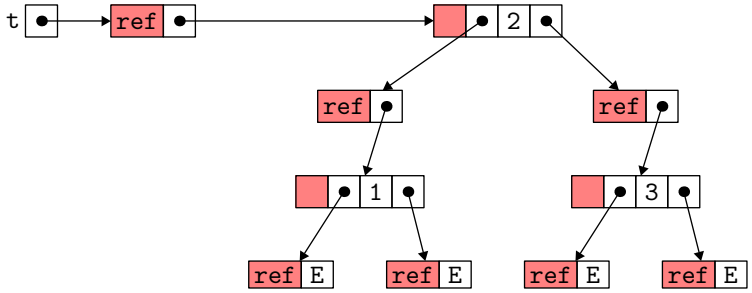
Une autre solution encore consiste à utiliser un type algébrique, avec un constructeur `E` pour l'arbre vide et un constructeur `N` pour un nœud, et à représenter les deux sous-arbres par des *références* sur des arbres.

```
type node = E | N of t * int * t
and  t    = node ref
```

On peut construire un arbre `t` contenant trois nœuds avec les déclarations suivantes

```
let t1 = ref (N (ref E, 1, ref E))
let t3 = ref (N (ref E, 3, ref E))
let t  = ref (N (t1,    2, t3   ))
```

et on peut le représenter graphiquement ainsi :



Il est important de comprendre que les quatre feuilles `ref E` ci-dessus ne peuvent pas être partagées, à la différence de la valeur `null` dans la section 2.1. En effet, ce sont elles qui doivent permettre de modifier les sous-arbres gauche et droit des nœuds `1` et `3`, de manière indépendante. L'occupation mémoire total est donc de $8N$ mots pour un arbre contenant N nœuds : 6 mots pour chacun des N nœud, et 2 mots pour chacune des $N+1$ feuilles. Si on omet la référence au sommet, pour être cohérent avec les sections précédentes, cela fait $8N$ mots au total.

Cette solution est certes encore plus gourmande que la précédente, mais elle a de multiples avantages. En premier lieu, elle conserve l'élégance du type algébrique. C'était déjà en partie le cas de la solution précédente, avec le type `option`, mais ici on a d'une part un constructeur spécifique pour l'arbre vide, et d'autre part un filtrage plus simple sur un arbre non vide (pas d'enregistrement). Mais surtout, elle permet une programmation en « passage par référence » particulièrement élégante [3]. Ainsi on peut écrire la fonction qui insère un élément `x` dans un arbre `t` aussi simplement que

```
let rec add x t = match !t with
  | E           -> t := N (ref E, x, ref E)
  | N (l, v, r) -> if x < v then add x l else if x > v then add x r
```

Il n'y a plus lieu de modifier les sous-arbres gauche et droit en remontant, puisque la modification a pu être faite tout en bas de l'arbre, en modifiant la référence qui représentait la feuille au point d'insertion. On ne dénature pas le code Java original, en conservant sa structure récursive. Dans un langage avec un passage par référence natif, comme `C++`, on aurait probablement écrit quelque chose de très analogue, du genre

```
void add(Tree &t, int x) {
  if (t == NULL) t = new_tree(NULL, x, NULL);
  else if (x < t->elt) add(t->left, x);
  else if (x > t->elt) add(t->right, x);
}
```

Les autres fonctions qui opèrent sur les arbres profitent de la même façon de cette possibilité de passage par référence. Ainsi la fonction qui supprime le plus petit élément dans un arbre non vide s'écrit ainsi :

```
let rec remove_min t = match !t with
  | E           -> assert false
  | N (l, _, r) -> if !l = E then t := !r else remove_min l
```

3. Bien entendu, OCaml ne connaît que le passage par valeur ; il s'agit ici du passage par valeur d'une référence, c'est-à-dire du pointeur vers le bloc représentant cette référence.

Un autre avantage de cette approche est qu'il n'y a tout simplement *aucune encapsulation* à effectuer. Le type `t` ci-dessus, ou encore la fonction `add: int -> t -> unit` que nous avons donnée, sont directement ceux attendus par l'interface `SET`. Au final, le code est plus court que pour les deux solutions précédentes.

2.4. Avec un type algébrique mutable

Pour cette quatrième et dernière solution, on imagine une variante du langage OCaml qui autorise de déclarer un argument de constructeur comme étant mutable[4]. Après tout, la représentation en mémoire d'un constructeur non constant est un bloc et rien n'exclut, si ce n'est le typage, d'en modifier les champs. Ainsi on peut imaginer une hypothétique déclaration de la forme

```
type tree = E | N of mutable tree * int * mutable tree
```

pour signifier que le premier et le troisième arguments du constructeur `N` peuvent être modifiés. On peut imaginer ensuite que le filtrage donne accès aux valeurs gauches correspondantes, pour les passer en arguments à la construction g `<-` e que l'on utilise déjà pour la modification des champs d'enregistrements. Ainsi on pourrait écrire la fonction `delete_min` qui supprime le plus petit élément d'un arbre non vide de la façon suivante :

```
let rec delete_min t = match t with
  | E           -> assert false
  | N (E, _, r) -> r
  | N (l, _, _) -> l <- delete_min l; t
```

Pour tester une telle solution, il y a plus simple que de modifier le langage OCaml : il suffit de se donner ici deux fonctions `set_left` et `set_right` pour modifier respectivement le premier et le troisième argument d'un arbre de la forme `N(l,v,r)`. On peut les écrire avec le module `Obj` de la bibliothèque OCaml, qui contourne le système de types.

```
let set_left  (x: tree) (t: tree) = Obj.set_field (Obj.repr x) 0 (Obj.repr t)
let set_right (x: tree) (t: tree) = Obj.set_field (Obj.repr x) 2 (Obj.repr t)
```

Les types explicites sont ici importants. Sans eux, `set_left` et `set_right` seraient polymorphes et on aurait vite fait de se tromper, par exemple en confondant une valeur et un arbre. Par ailleurs, on pourrait écrire des versions un peu plus défensives de ces deux fonctions, qui commencent par vérifier que le premier argument `x` est bien un bloc de taille 2.

Une telle solution présente tous les avantages du type algébrique, en termes d'élégance et d'occupation mémoire. Ainsi on construit un arbre aussi facilement que

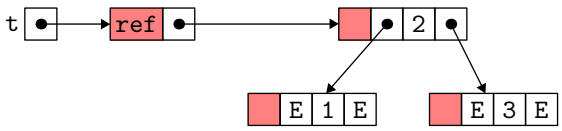
```
let t = N (N (E, 1, E), 2, (E, 3, E))
```

et sa représentation en mémoire n'est pas différente de celle d'un arbre immuable. Comme on l'a déjà fait plusieurs fois, on obtient la signature `SET` attendue en encapsulant un tel arbre dans une référence.

```
module S : SET = struct
  type t = tree ref
  let create () = ref E
  let add x s = s := add x !s
  ...
```

Ainsi l'arbre `t` qui nous sert d'illustration depuis le début est aussi simple que

4. Une telle possibilité existe dans le langage Caml Light.



L'occupation mémoire est de nouveau $4N$ mots pour un arbre contenant N nœuds, comme pour la solution purement applicative ou la solution avec `null` de la section 2.1.

3. Évaluation expérimentale

Dans cette section, on compare les performances des différentes solutions présentées ci-dessus. On les dénomme respectivement par « null » (section 2.1), « option » (section 2.2), « alg ref » (section 2.3) et « alg mut » (section 2.4). Pour être complet, on les compare également avec la solution consistant en une référence contenant un arbre purement applicatif, dénommée « pure ref » ci-dessous. Cette dernière solution ne constitue pas à proprement parler des arbres mutables, mais il reste intéressant d'en évaluer les performances.

Le protocole de test est le suivant. On évalue les opérations d'ajout, de recherche et de suppression dans différents cas de figure, en se livrant aux trois expériences suivantes (les noms des tests sont donnés entre parenthèses, pour être utilisés ensuite) :

1. On commence par construire un arbre en insérant tous les entiers de 0 à 9 999 dans un arbre initialement vide, dans cet ordre (add-seq). On a donc construit un peigne. Puis on recherche dans cet arbre les valeurs de -10 000 à 9 999 (mem-seq).
2. On construit ensuite un second arbre, en insérant un million de valeurs aléatoires dans un arbre initialement vide (add-rnd). Puis, de même, on recherche dans cet arbre les valeurs de -10 000 à 9 999 (mem-rnd). Enfin, on retire un à un tous les éléments de cet arbre, dans l'ordre où ils avaient été insérés (rmv-rnd).
3. On construit enfin un troisième arbre, toujours en insérant un million de valeurs aléatoires dans un arbre initialement vide mais cette fois en ne tirant que parmi 20 000 valeurs possibles (add-rnd2). Il y a donc une majorité de cas où l'élément se trouve déjà dans l'arbre. On retire alors un à un tous les éléments de cet arbre, en suivant exactement le même tirage aléatoire que pour leur insertion (rmv-rnd2). Cette fois, on a donc une majorité de cas où l'élément ne se trouve plus dans l'arbre.

Pour chaque test, on effectue cinq mesures du temps de calcul, on élimine la plus petite et la plus grande et on fait la moyenne des trois restantes. Le générateur aléatoire d'OCaml est réinitialisé de la même façon avant chaque mesure. Les résultats (en secondes, mesurés sur un processeur AMD X86-64 sous Linux) sont donnés dans cette table :

	expérience 1		expérience 2			expérience 3	
	add-seq	mem-seq	add-rnd	mem-rnd	rmv-rnd	add-rnd2	rmv-rnd2
`null` (2.1)	1.320	0.596	*1.352	0.503	*1.267	0.427	0.023
option (2.2)	2.168	0.695	4.339	0.523	2.579	1.067	0.028
alg ref (2.3)	*0.611	0.640	1.943	0.512	1.548	*0.324	*0.012
alg mut (2.4)	1.340	0.576	1.436	0.499	1.347	0.451	0.020
pure ref	3.002	*0.568	3.590	*0.497	2.930	1.065	0.021

On note que la solution utilisant un hypothétique type algébrique mutable (section 2.4) offre des performances comparables à celle de la solution utilisant un pointeur nul (section 2.1). Vu l'intérêt indéniable du type algébrique par rapport au pointeur nul, on pourrait regretter qu'OCaml n'offre pas cette possibilité de déclarer certains arguments de constructeurs comme étant mutables.

Mais on constate ensuite que la solution qui l'emporte le plus souvent est celle utilisant des références (section 2.3). Lorsqu'elle ne l'emporte pas, elle reste respectivement à 12% (mem-seq), 44% (add-rnd), 3 % (mem-rnd) et 22% (rmv-rnd) de la meilleure solution. D'autre part, on note qu'elle est bien meilleure que la solution utilisant un type `option`. Ce n'était pas évident à priori, les représentations mémoire étant très proches.

Pour être complet, on rappelle l'occupation mémoire des diverses solutions, en nombre de mots mémoire pour un arbre contenant N nœuds, et on donne le nombre de lignes de code de chaque solution (pour un module complet d'interface `SET`).

	`null`	option	alg ref	alg mut	pure ref
occupation mémoire	$4N$	$6N$	$8N$	$4N$	$4N$
lignes de code	36	42	29	45	34

La solution utilisant des références est certes la plus gourmande (deux fois plus que les moins coûteuses) mais aussi, inversement, la plus économe en lignes de code, comme nous l'avions fait remarquer section 2.3.

4. Conclusion

Réaliser des arbres mutables en OCaml s'est avéré plus subtil que de prime abord, surtout si on cherche à conserver les avantages des types algébriques, à savoir le typage statique et le filtrage. On a présenté plusieurs solutions. On peut définitivement écarter la solution utilisant le type `option`, qui ne présente quasiment aucun avantage. La solution utilisant des références, en revanche, se dégage comme la plus élégante — on conserve notamment un type algébrique — mais aussi comme l'une des plus efficaces, car elle permet de programmer dans un style « passage par référence ». Elle est certes relativement coûteuse en mémoire (deux fois plus que pour un arbre pur) mais cela reste très raisonnable.

Même si cela est assez évident, il convient de noter que tout ce qui a été présenté dans cet article reste valable si on cherche à équilibrer les arbres, que ce soit des AVL comme dans la bibliothèque OCaml, des arbres rouges et noirs ou encore toute autre solution.

Remerciements. Je remercie Xavier Leroy pour m'avoir suggéré l'approche « passage par référence ». Toutes les figures de cet article ont été réalisées avec MLPOST [1].

Références

[1] R. Bardou, J.-C. Filliâtre, J. Kanig, and S. Lescuyer. Faire bonne figure avec Mlpost. In *Vingtièmes Journées Francophones des Langages Applicatifs*, Saint-Quentin sur Isère, Jan. 2009. INRIA.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[3] D. E. Knuth. *The Art of Computer Programming, volumes 1–4A*. Addison Wesley Professional, 1997.

[4] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[5] R. Sedgewick and K. Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.