Structures de Données (1)

Introduction

Nous allons maintenant aborder un domaine fondamental de l'informatique : les *structures de données*. De manière générale, une structure de donnée est l'association d'un ou plusieurs types, servant à contenir des données (des valeurs d'un autre type, en général), et d'algorithmes servant à manipuler ces types.
Nous avons déjà vu plusieurs exemples de structures de données simples : tableaux et listes, qui servent à stocker une collection d'élément dans un ordre précis, à traverser cette collection et, respectivement, à accéder à un élément par son indice ou à ajouter un élément au début d'une liste. Les références, d'une certaine manière, forment une structure très limitée ne pouvant contenir qu'une seule valeur.
De manière générale, on définit une *interface*, ou *spécification*, d'une structure de données. En Caml cela se fait souvent par un fichier `.mli` (« i »pour « interface ») [1]. Par exemple, une abstraction de la notion de tableau pourrait avoir la signature suivante (**value** déclare le type d'un identifiant, dans une interface) :

```
type 'a vect (* tableau d'elements de type 'a *)

value make : int -> 'a -> 'a t  (* creation *)
value get : 'a t -> int -> 'a  (* acces *)
value set : 'a t -> int -> 'a -> unit (* modification *)
value length : 'a t -> int   (* longueur *)
```

La spécification permet de travailler avec une structure de données sans savoir exactement comment elle fonctionne. Caml est d'ailleurs fourni avec plusieurs modules standards, et Python propose une large collection de structures de données (ensembles, tableaux, piles, etc.). Pour une même spécification, plusieurs *implémentations* concrètes peuvent exister. Cette notion de spécification et implémentation existe aussi pour les algorithmes en général, nous avons par exemple vu que plusieurs algorithmes de tri existent (leur spécification, « trier la liste qui leur est passée en argument », étant commune).

Structure Impérative, Structure Persistante

On distingue deux grandes familles de structures de données : les structures *impératives* et les structures *persistantes*. Les premières fournissent des opéra-

1. En OCaml, un système très puissant de modules donne de meilleures capacités d'abstraction.

tions qui modifient en place la structure ; par exemple, l'opération `a.(i) <- x` qui assigne la valeur `x` à la $i^{\text{ème}}$ case d'un tableau, modifie le tableau en place et l'ancienne valeur est perdue. Inversement, dans le cas *persistant*, chaque structure est dite *immutable* (on ne peut pas la modifier) et les opérations renvoient de nouvelles structures au lieu de les modifier. C'est notamment le cas des listes, nous avons vu que `rev` ou `@` (la concaténation) ne modifient jamais leurs arguments, mais renvoient de nouvelles listes à la place.
Cette distinction est généralisable : Wikipédia, par exemple, fournit un historique des modifications, ce qui permet de « revenir en arrière », pour accéder à une ancienne version qui n'est donc pas perdue. De même, un livre de comptes ou une feuille de salaire [2] ne sont pas effacés et réécrits chaque années mais minutieusement archivés.
Nous allons étudier, pour chaque structure de données, au moins une version impérative et une version persistante. Nous verrons les piles, files d'attente, files de priorité et conteneurs associatifs (« map »).

1 Digression : Représentation Mémoire

Avant toute chose, clarifions certaines propriétés de la représentation des valeurs en Caml. Comme presque tous les langages de programmation, Caml utilise la notion de « pointeur » ou *référence* pour inclure certaines valeurs dans d'autres. En pratique, sans trop rentrer dans les détails, chaque valeur a une *adresse* mémoire unique (comme une adresse postale), et un pointeur est simplement cette adresse. En particulier, les tableaux Caml sont toujours manipulés à travers leur adresse. La figure 1 montre la structure mémoire de tableaux de tableaux créés ainsi :

```
let a = init_vect 2 (fun i -> init_vect 2 (fun j -> i+10*j));;
let b = make_vect 2 (init_vect 2 (fun i -> i+100));;
let b' = b;;
```

On remarque la différence entre `init_vect` et `make_vect`, le second utilisant la même valeur dans chaque case du tableau. Il faut donc bien prendre soin d'utiliser `init_vect` pour construire des matrices. Les lignes pointillées indiquent que `a` et `b` ne sont que des *noms* pour les deux tableaux, et non des valeurs mémoire. Chaque tableau est préfixé par son adresse mémoire (choisie au hasard) pour plus de clarté et consiste en une suite contigüe de cases mémoire contenant les éléments du tableau. On peut tester si des valeurs ont la même adresse à l'aide de l'opérateur == (ne pas confondre avec le = habituel), comme l'exemple ci-après le montre [3].

```
# let a = make_vect 2 0;;
# let b = make_vect 2 0;;
# a = b;;
- : bool = true
```

2. Ne vous inquiétez pas, ces mystérieuses notions vous concerneront un jour...
3. J'utilise une fonction "magique" pour obtenir les adresses des valeurs, mais des raisons éthiques m'interdisent de divulguer laquelle.

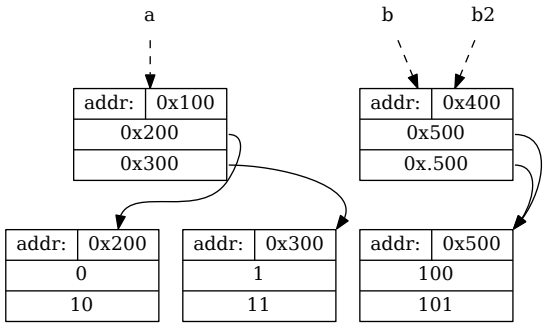


FIGURE 1 – Tableaux de tableaux

```
# a == b;;
- : bool = false
# printf__sprintf "adresses: a: %x, b: %x" (addr_of a) (addr_of b);;
- : string = "adresses: a: 3fc564a70cf8, b: 3fc564a70cd0"
```

La figure 2 montre la disposition mémoire des listes générées par le code suivant. Chaque nœud de liste créé par le constructeur :: occupe sa propre place mémoire (avec l'élément puis l'adresse du nœud suivant), et la liste vide [] est à l'adresse 0.

```
# let l1 = [1;2;3];;
# let l2 = 10 :: l1;;
# let l3 = match l1 with
    | [] -> failwith "impossible"
    | _ :: l1' -> 42 :: l1;;
```

2 Arbres

2.1 Définitions et Exemple Simple

Comme nous avons vu, Caml fournit des *types algébriques* dont nous avons vu plusieurs exemples (listes et options). Ces types permettent plus généralement de représenter des *arbres*. Un arbre est une structure récursive composée de nœuds, qui ont des *sous-arbres* comme enfants, et de feuilles (nœuds sans enfants). Chaque arbre non vide possède une *racine* qui est le seul nœud qui n'est enfant d'aucun autre nœud. On parle d'arbre *binaire* si chaque nœud a au plus deux enfants. Par exemple, la figure 3 regroupe deux représentations d'un arbre :

— la figure 3a contient la représentation mémoire de l'arbre, avec l'arbre vide représenté par le symbole $\perp$ (voir le code ci-après) ;

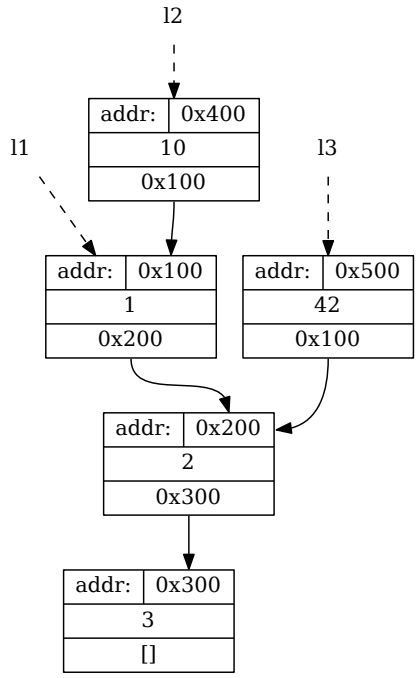


FIGURE 2 – Listes

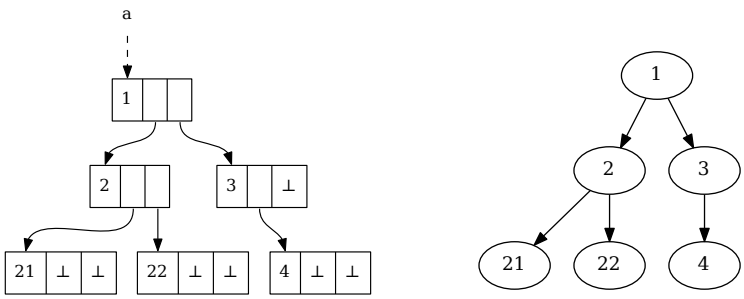
— la seconde figure 3b représente l'arbre de manière indépendante de sa représentation en Caml.

Cet arbre, dont les nœuds sont étiquetés dans $\{1, 2, 3\}$, les feuilles dans $\{21, 22, 4\}$, et dont la racine est le nœud étiqueté 1, peut être construit par le code suivant en prenant comme convention qu'une feuille est un nœud dont les enfants sont l'arbre Vide :

```
type arbre =
    | Noeud of int * arbre * arbre
    | Vide;;

let a = Noeud (1,
    Noeud (2, Noeud (21, Vide, Vide), Noeud (22, Vide, Vide)),
    Noeud (3, Noeud (4, Vide, Vide), Vide));;
```

Exercice : en partant de la définition du type arbre au dessus, écrire une fonction sum: arbre -> int qui calcule la somme des étiquettes de tous les nœuds. Justifier sa terminaison.



(a) Représentation Mémoire (b) Arbre Platonique

FIGURE 3 – Arbres d'entiers

Exercice : écrire une fonction qui « renverse » un arbre, c'est-à-dire qui appliquer une symétrie verticale à son argument.
Exercice : écrire une fonction `map: (int -> int) -> arbre -> arbre` qui applique la fonction passée en argument à toutes les étiquettes des nœuds de l'arbre.

2.2 Propriétés Basiques

La profondeur d'une feuille est la distance entre la racine et cette feuille (elle est donc de 0 si la racine n'a pas d'enfants). La *hauteur* h d'un arbre est la distance maximale entre la racine et la feuille la plus profonde, c'est-à-dire le maximum de la profondeur des feuilles, définie récursivement par :

$$h(t) = \begin{cases} 0 & \text{si } t \text{ est une feuille} \\ 1 + \max_{i \in I} h(t_i) & \text{si } h \text{ a pour enfants les } (t_i)_{i \in I} \end{cases}$$

Pour un arbre binaire on peut prouver que $n \leq 2^{h+1} - 1$, avec n le nombre de nœuds (feuilles comprises) et h la hauteur de l'arbre. La preuve se fait par récurrence forte sur la hauteur de l'arbre :

- la propriété est vraie pour tout arbre de profondeur 0 (un seul nœud) ;
- si c'est vrai pour tout arbre de profondeur $h' \leq h$, alors tout arbre binaire de profondeur $h+1$ est nécessairement un nœud possédant deux enfants t_1 et t_2 de profondeur au plus h. Par hypothèse d'induction $n_1 \leq 2^{h+1} - 1$ et $n_2 \leq 2^{h+1} - 1$ avec n_i le nombre de nœuds de h_i ; on a donc $n = 1 + n_1 + n_2 \leq 1 + 2^{h+1} - 1 + 2^{h+1} - 1$, soit $n \leq 2 \cdot 2^{h+1} - 1$, soit $n \leq 2^{(h+1)+1} - 1$.

Un arbre binaire de profondeur h comportant $2^{h+1} - 1$ nœuds est dit *arbre binaire complet*. De tels arbres ont une structure très symétriques car tous les chemins de la racine vers les feuilles ont la même longueur.

On dit qu'un arbre binaire est *équilibré* si la différence de profondeur entre ses feuilles est petite (dans le cas strict, au plus de 1, mais on peut relaxer un peu cette contrainte).

2.3 Arbre de Recherche

On suppose que les arbres possèdent une fonction N qui à chaque nœud (ou feuille) associe une étiquette dans un ensemble totalement ordonné.
Un arbre binaire est dit *arbre de recherche* si, pour tout nœud t, tous les nœuds t_1 du sous-arbre gauche de t satisfont $N(t_1) < N(t)$, et tous les nœuds t_2 du sous-arbre gauche de t satisfont $N(t_2) > N(t)$. La figure 4 présente un exemple d'arbre binaire de recherche.

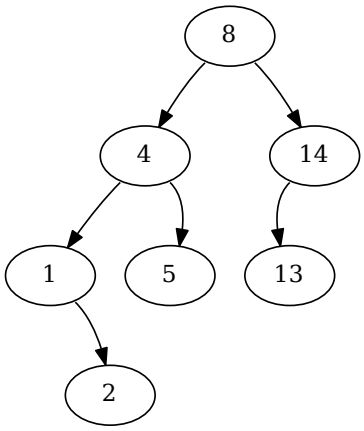


FIGURE 4 – Arbre Binaire de Recherche

Les arbres de recherche sont ainsi nommés car la recherche d'éléments y est facile. Il suffit de comparer récursivement l'élément qu'on cherche avec l'étiquette d'un nœud, puis de traiter récursivement le sous-arbre gauche ou droit en fonction du résultat. Le code suivant implémente cet algorithme (la fonction `search : 'a -> 'a tree -> bool` renvoie `true` ssi l'élément recherché est dans l'arbre) :

```
type 'a tree =
    | Node of 'a * 'a tree * 'a tree
    | Empty;;

(* recherche si x est dans t *)
let rec search x t = match t with
    | Empty -> false
    | Node (y, l, r) ->
```

```
    if x < y then search x l
    else if x > y then search x r
    else true;;
```

Dans le cas d'arbres *équilibrés*, cette recherche se fait en temps $O(ln(n))$ avec n le nombre de nœuds (ou $O(h)$ avec h la hauteur), ce qui est efficace (comparable à la recherche *dichotomique* dans un tableau). Vous verrez en seconde année comment insérer ou enlever des nœuds dans un arbre binaire de recherche équilibré, en maintenant l'équilibre (pour maintenir cette complexité logarithmique fort utile). Par exemple les arbres AVL maintiennent une différence de hauteur entre sous-arbres gauche et droit de au plus 1. Dans un arbre équilibré, on peut borner la hauteur de l'arbre (profondeur maximale d'une feuille) par $O(log_2(n))$. Sachant que search parcourt, au pire cas, exactement un chemin de la racine à une feuille, le nombre d'appels récursifs (et de comparaison) est donc borné par $O(log_2(n))$.
Cette structure de données est encore une instance du paradigme « diviser pour régner » : à chaque appel récursif, l'ensemble des nœuds candidats (parmi lesquels on cherche l'élément) est divisé environ par 2, ce qui explique son efficacité.

2.4 Arbres Représentant des Expressions

Caml descend de la famille de langages ML qui fut conçue comme langage de preuve. La puissance du filtrage et des types algébriques rend les langages ML particulièrement adaptés à la manipulation symbolique d'expressions (compilation, preuve, calcul symbolique, etc.).
Voyons un exemple simple : des expressions arithmétiques basiques sur les entiers, décrites par le type suivant :

```
type expr =
    | Const of int  (* constante *)
    | Plus of expr * expr (* x plus y *)
    | Fois of expr * expr (* x fois y *)
    | Moins of expr  (* -x *)
```

Exercice : écrire une fonction eval : expr −> int qui calcule récursivement la valeur d'une telle expression.
On peut étendre ce type pour permettre de définir des variables dans le langage arithmétique lui-même, en ajoutant un constructeur Var qui représente une variable (avec un nom pour différencier les variables entre elles).

```
type expr =
    | ...
    | Var of string
```

Exercice : Écrire une fonction d'évaluation eval2 qui calcule la valeur d'une expression à partir d'une *fonction de valuation* associant à chaque variable sa valeur entière. Cette fonction a donc pour type expr −> (string −> int) −> int. On aura donc :

```
(* x.y - 10.x *)
let e = Plus (Fois (Var "x", Var "y"), Moins (Fois (Var "x", Const 10)));;
```

```
let valuation = function
    | "x" -> 5
    | "y" -> 1
    | _ -> 0;;

eval2 e valuation = -45;;
```

Exercice : (plus avancé) écrire un petit programme de *dérivation symbolique*. Comme vos calculatrices, un tel programme prend en entrée un arbre représentant une expression mathématique (avec variables, et des fonctions « classiques » telles que $\cos x$ ou $\sin x$, et le nom d'une variable, et renvoie un arbre représentant la dérivée de la fonction par rapport à cette variable.
La spécification de cette exercice est donc (on peut ajouter des constructeurs d'expressions) :

```
type expr =
    | Var of string  (* nom de variable *)
    | Const of int  (* constante *)
    | Plus of expr * expr (* x plus y *)
    | Fois of expr * expr (* x fois y *)
    | Moins of expr  (* -x *)
    | Cos of expr  (* cos x *)
    | Sin of expr  (* sin x *)
    | Puissance of expr * int  (* x puissance n *)

value deriver : expr -> string -> expr

(* d(5.x + x.x)/dx = 5 + 2x *)
deriver (Plus (Fois (Var "x", Const 5), Fois (Var "x", Var "x"))) "x" =
    Plus (Const 5, Plus (Var "x", Var "x"));;
```

3 Piles

Nous allons spécifier et utiliser des *piles*[4], une structure classique étroitement liée à la récursion.

3.1 Spécification

La spécification d'une structure impérative de pile est la suivante (une pile fonctionnelle est tout simplement une liste, avec un peu de filtrage pour accéder au premier élément) :

```
type 'a stack   (* pile contenant des valeurs de type 'a *)
exception EmptyStack

value create : unit -> 'a t  (* nouvelle pile *)
value is_empty : 'a t -> bool (* pile vide? *)
value top : 'a t -> 'a  (* sommet de la pile *)
value pop : 'a t -> 'a  (* enlever sommet de la pile *)
value push : 'a t -> 'a -> unit (* pousser un element au sommet *)
```

4. aussi appelées « LIFO » en anglais, pour « last-in, first-out » qui décrit leur comportement.

En imaginant la pile comme une pile de livres (ou d'assiettes si vous aimez casser la vaisselle), les opérations sont assez intuitives. On choisit cette fois-ci de *lever une exception* si une opération de lecture est effectuée sur une pile vide plutôt que renvoyer une exception.

- create : crée une nouvelle pile ;
- is_empty : renvoie true si la pile ne contient aucun élément ;
- top : renvoie l'élément (le livre) au sommet de la pile (ou échoue en lançant EmptyStack si la pile est vide) ;
- pop : retourne et enlève l'élément du sommet de la pile (modifie donc la pile) ;
- push : ajoute un nouvel élément au sommet de la pile.

3.2 Implémentation à base de Liste

La première implémentation, très simple, s'appuie sur la structure de liste avec une référence. Toutes les opérations se font trivialement en temps constant.

```
type 'a stack == 'a list ref;;
exception EmptyStack;;

let create () = ref [];;

let is_empty stack = !stack = [];;

let top stack = match !stack with
    | [] -> raise EmptyStack
    | x::_ -> x ;;

let pop stack = match !stack with
    | [] -> raise EmptyStack
    | x::l' ->
        stack := l';
        x ;;

let push stack x = stack := x :: !stack;;
```

3.3 Implémentation à base de Tableau

On considère maintenant une pile dont les éléments sont stockés dans un tableau. Par soucis de simplicité[5] va un peu modifier la spécification de deux manières :

1. chaque pile a une hauteur maximale, et tout dépassement lance l'exception StackOverflow
2. on fournit un élément de « remplissage » à la création de la pile (donc create: 'a -> 'a t).

5. et de respect du programme...

```
type 'a stack = {
    mutable size : int;
    tab : 'a vect;
} ;;

exception EmptyStack;;
exception StackOverflow;;

let create x = {
    tab = make_vect 100 x;
    size = 0;
};;

let is_empty stack = stack.size = 0;;

let top stack =
    if is_empty stack then raise EmptyStack;
    stack.tab.(stack.size - 1);;

let pop stack =
    if is_empty stack then raise EmptyStack;
    let x = stack.tab.(stack.size - 1) in
    stack.size <- stack.size - 1;
    x ;;

let push stack x =
    if vect_length stack.tab = stack.size then raise StackOverflow;
    stack.tab.(stack.size) <- x;
    stack.size <- stack.size + 1;;
```

Cette structure de pile peut être étendue, au prix de la simplicité, pour respecter la spécification – l'astuce est de rendre le tableau re-dimmensionnable. On obtient quelque chose de proche des tableaux de Python.
Il est bon de noter que l'évaluation concrète des fonctions récursives de Caml repose sur l'utilisation d'une pile, dite *pile d'exécution*. Cette pile sert à stocker les paramètres des fonctions[6] ; chaque appel récursif pousse de nouvelles informations sur la pile, et chaque fonction enlève ses arguments de la pile avant de retourner son résultat. C'est pour cela qu'évaluer le code suivant va planter Caml (la pile se remplit en entier... D'où l'importance de prouver la terminaison de son code). On parle de *dépassement de pile* ou *stack overflow*[7].

```
let rec infinie x = 1 + infinie x;;
```

Cependant, les *appels récursifs terminaux* (voir cours précédents) peuvent réutiliser la même portion de la pile et donc éviter certains cas de *stack overflow*.

6. ainsi que d'autres informations nécessaires comme le pointeur de code, etc.
7. cette expression imagée provient sûrement de ce qui arrive quand la pile de vaisselle sale fait déborder l'évier et provoque une inondation dans l'appartement.

4 Application : Évaluation Postfixe Simple

On peut utiliser les piles pour évaluer un langage arithmétique simple écrit dans une *notation postfixe*[8] (présent dans certaines vieilles calculatrices de chez TI par exemple). Si on suppose que l'on connait l'arité (le nombre d'arguments) de chaque symbole, il est très facile de parser (lire) des expressions arithmétiques dans un tel langage. Par exemple, $2\ 3\ +\ 5\ *\ 2\ -$ dénote sans ambigüité l'expression $((2+3)*5)-2$.
Évaluer une telle expression se fait très simplement à partir d'une pile, comme le montre le code suivant :

```
type operation =
  | OpConst of int (* constante *)
  | OpPlus         (* addition *)
  | OpFois         (* multiplication *)
  | OpSoustraire   (* soustraction *)
;;

let eval l =
  let stack = create () in
  (* fonction auxiliaire qui traverse la liste *)
  let rec aux l = match l with
      | [] -> pop stack
      | op :: l' ->
        begin match op with
        | OpConst i -> push stack i
        | OpPlus ->
            let a = pop stack in
            let b = pop stack in
            push stack (a+b)
        | OpFois ->
            let a = pop stack in
            let b = pop stack in
            push stack (a*b)
        | OpSoustraire ->
            let a = pop stack in
            let b = pop stack in
            push stack (b-a)  (* attention *)
        end;
        aux l'  (* evaluer la suite *)
  in
  aux l;;

eval [OpConst 2; OpConst 3; OpPlus; OpConst 5; OpFois;
      OpConst 2; OpSoustraire] = 23;;
```

Exercice : Écrire une fonction d'évaluation d'une telle liste postfixe d'opérations en utilisant la récursion, sans boucle ni constructions impératives.
Exercice : Écrire une fonction qui prend une liste d'opérations en forme postfixe, et construit l'arbre de syntaxe (de type expr) correspondant. En déduire une fonction d'évaluation des listes d'opérations.

8. Aussi appelée « Notation Polonaise Inverse », voir `https://fr.wikipedia.org/wiki/Notation_polonaise_inverse`.

Exercice : (difficile) ajouter au type operation des variants get et set représentant respectivement l'accès à une variable (unaire) et la modification d'une variable (binaire) par son nom, et modifier la fonction d'évaluation pour en tenir compte. Il est nécessaire de stocker quelque part la valeur actuelle des variables pendant l'évaluation (par exemple dans une liste d'association).
Exemple : la liste d'opérations `1 "a" set 2 "a" get 3 + *` représente l'expression $a := 1;\ 2 \cdot (a + 3)$. Il suffirait maintenant d'une construction de boucle (ou de récursion) pour obtenir un langage de programmation complet [9] !
Exercice : (difficile) écrire un *parseur* qui prend une chaîne de caractères en entrée (de type string) et renvoie une liste d'opérations enveloppée dans Some, ou None si la chaîne est invalide.

9. Si vous arrivez jusque là, la conception d'une construction de boucle/récursion pour ce langage est très instructive. N'hésitez pas à me demander...