Itérer avec persistance

Jean-Christophe Filliâtre[1]

1: CNRS – Laboratoire de Recherche en Informatique
Bâtiment 490, Université Paris Sud
91405 ORSAY Cedex, France
filliatr@lri.fr

Résumé

L'énumération des éléments d'une structure de données est généralement réalisée en ML par l'intermédiaire d'une fonction d'ordre supérieur. Cet article présente une alternative, sous la forme d'itérateurs pas à pas, à l'instar de ce qui se fait en programmation orientée objets, mais basés sur des structures persistantes, de manière à permettre notamment un éventuel *backtracking*. Plusieurs façons de parcourir les arbres binaires sont examinées, et des liens étroits avec le *zipper* de Gérard Huet sont établis.

1. Introduction

Le programmeur ML est habitué à énumérer les éléments d'une structure de données à l'aide de fonctions d'ordre supérieur. Ainsi, une structure de données réalisée par un type `t` et représentant un ensemble d'éléments d'un certain type `elt` sera naturellement équipée d'une fonction de la forme[1]

```
fold : (elt → α → α) → t → α → α
```

dont l'effet est de construire une valeur de type α en partant d'une valeur initiale (son troisième argument) et en appliquant une fonction (son premier argument) à chaque élément de l'ensemble (son deuxième argument) et à la valeur en cours de construction. S'il s'agit par exemple d'un ensemble d'entiers, et que le type `elt` est alors `int`, on peut ainsi faire la somme de tous les éléments d'un ensemble `s` aussi simplement que par

```
fold (fun x n → n+x) s 0
```

Lorsque la fonction passée en argument n'est utilisée que pour ses effets, on utilise une forme dégénérée de la fonction `fold` présentant le type suivant :

```
iter : (elt → unit) → t → unit
```

On peut ainsi afficher tous les éléments de notre ensemble `s` aussi simplement que par `iter (fun x → Printf.printf "%d\n" x) s`. De telles fonctions d'ordre supérieur parcourant les éléments d'une structure de données sont appelées des *itérateurs* et leur utilisation constitue probablement le trait le plus idiomatique de la programmation fonctionnelle. Le programmeur ML expérimenté les utilise cent fois par jour, apprécie leur concision et leur clarté, et n'imagine pas de moyen plus élégant de procéder.

[1]Cet article est illustré par du code écrit dans le langage Objective Caml [1] (abrégé en Ocaml par la suite), mais s'adapte aisément dans n'importe quel autre langage de la famille ML.

Il est cependant des situations, certes rares, où l'utilisation de tels itérateurs se révèle peu commode, voire tout simplement impossible. Si le type de la structure de données en question est concret, il suffira de la parcourir d'une manière différente, plus immédiate. Mais si type en question est *abstrait*, et que la seule manière d'en parcourir les éléments est précisément la donnée de ces itérateurs, alors le programmeur peut se retrouver dans une situation où il n'existe plus de moyen simple ou efficace de procéder.

Le premier exemple est celui d'une énumération qu'on souhaite interrompre prématurément. Ainsi, si l'on cherche à déterminer s'il existe dans notre ensemble `s` un élément vérifiant une certaine propriété `p : int → bool`, alors la solution consistant à écrire

```
fold (fun x b → p x || b) s false
```

n'est pas satisfaisante dans le contexte d'un langage strict comme OCAML, car la totalité des éléments sera forcément examinée, même si l'on rencontre rapidement un élément vérifiant `p` et malgré le caractère paresseux de l'opérateur `||` (car appliqué ici à une valeur et non à une expression). Une solution possible consiste à utiliser alors une exception pour interrompre l'itération. On peut ainsi réécrire efficacement notre recherche en levant l'exception prédéfinie `Exit` dès qu'un élément vérifiant `p` est rencontré :

```
try iter (fun x → if p x then raise Exit) s; false with Exit → true
```

Bien entendu, une structure de données réalisant des ensembles fournira presque toujours une fonction `exists : (elt → bool) → t → bool` ayant exactement cet effet, mais on finit toujours par tomber sur une situation où la fonction de recherche qu'on souhaite réaliser n'est pas fournie de manière primitive. Le recours à une exception, même s'il est efficace, n'est pas toujours très facile à mettre en œuvre — lorsqu'une valeur doit être retournée, il faut soit définir une exception particulière, soit recourir à une référence — et ne constitue pas un code très élégant.

C'est là qu'un programmeur JAVA ou C++ pourra mettre en avant l'avantage de sa manière de procéder. Dans ces langages-là, en effet, les itérateurs ne se présentent pas comme des fonctions d'ordre supérieur mais comme des structures de données à même de produire les éléments de l'énumération *un par un*. Ainsi en JAVA l'énumération de tous les éléments d'une structure de données `t` s'écrit de la manière typique suivante :

```
for (Iterator i = t.iterator(); i.hasNext(); ) {
  ... traiter i.next() ...
}
```

La méthode `iterator` de la structure de données construit un nouvel itérateur sur les éléments de `t`, puis les deux méthodes `hasNext` et `next` de cet itérateur indiquent respectivement s'il reste des éléments à énumérer et quel est l'élément en cours de visite le cas échéant. Il est fondamental de comprendre que l'itérateur est ici une structure *impérative* : l'appel à `next` renvoie l'élément en cours de visite *et* fait passer l'itérateur dans l'état suivant *par effet de bord* i.e. le positionne sur l'élément suivant dans l'énumération. Dans la suite, nous appelons *itérateur pas à pas* un tel itérateur.

S'il est vrai que, dans la plupart des cas, un tel itérateur serait plus lourd à utiliser que son homologue d'ordre supérieur, et beaucoup moins élégant par l'effet de bord caché dans la fonction retournant l'élément suivant, il n'en reste pas moins qu'il apporte une solution satisfaisante au problème de l'interruption prématurée de l'itération. En supposant donné en OCAML un itérateur *à la* JAVA `i` sur notre ensemble `s`, on peut aisément écrire notre fonction déterminant si un élément vérifie la propriété `p` :

```
let rec test () = has_next i && (p (next i) || test ())
```

Malheureusement, il existe (au moins) une deuxième situation dans laquelle ni les itérateurs comme fonctions d'ordre supérieur ni les itérateurs pas à pas à la JAVA n'offre de solution satisfaisante : lorsqu'on souhaite *revenir en arrière* dans l'énumération, autrement dit lorsque l'itérateur est impliqué dans un algorithme effectuant du *backtracking*. À titre d'exemple, supposons que nous souhaitions déterminer si notre ensemble d'entiers `s` contient un sous-ensemble dont la somme des éléments vaut 100. Si l'ensemble `s` est donné de manière concrète comme une liste d'entiers, il est très facile d'écrire un programme effectuant ce test[2] :

```
let rec somme sol = function
  | [] → sol = 100
  | x :: r → somme (sol+x) r || somme sol r
in
somme 0 s
```

Mais si l'ensemble `s` est réalisé par un type abstrait qui ne fournit que des itérateurs d'ordre supérieur ou à la JAVA, un tel algorithme basé sur le *backtracking* (essayer avec l'élément courant, puis essayer sans en cas d'échec) n'est plus possible. On pourrait imaginer construire la liste de tous les éléments grâce à l'itérateur dont on dispose, puis appliquer ensuite l'algorithme ci-dessus, mais cette solution est clairement inefficace en espace (car on construit une liste qui a la taille de la structure de données toute entière).

Il existe heureusement une meilleure solution. Elle consiste à définir un itérateur pas à pas réalisé par une structure de données *persistante*[3], c'est-à-dire où l'avancée à l'élément suivant ne modifie pas l'itérateur mais en retourne un nouveau. Supposons qu'on dispose d'un tel itérateur sous la forme d'un type `enum` et de deux fonctions `start` et `step` :

```
type enum
val start : t → enum
val step : enum → elt × enum
```

La fonction `start` construit un nouvel itérateur pour l'ensemble des éléments de la structure passée en argument ; on peut le voir comme pointant sur le « premier » élément de la structure. La fonction `step` retourne l'élément désigné par un itérateur ainsi qu'un *nouvel* itérateur pointant sur l'élément suivant. On suppose qu'elle lève l'exception prédéfinie `Exit` lorsque l'énumération est achevée. On peut alors réécrire ainsi l'algorithme ci-dessus procédant par *backtracking* :

```
let rec somme sol e =
  try
    let x,e = step e in somme (sol+x) e || somme sol e
  with Exit →
    sol = 100
in
somme 0 (start s)
```

Le code est en tout point similaire à celui opérant sur les listes, si ce n'est que ces dernières ont été remplacées par le type `enum` des énumérateurs. On remarque d'ailleurs que, dans le cas où le type abstrait `t` est en fait le type `int list`, il suffit de prendre `type enum = int list`, pour `start` la fonction identité, et pour `step` la fonction de déstructuration du constructeur `::`.

Dans cet article, nous nous intéressons à ces itérateurs pas à pas persistants, que nous appelons *réitérateurs*, puisque leur intérêt est dans le *backtracking*, c'est-à-dire dans le fait de *réitérer*. On pourrait d'ailleurs adopter le slogan suivant :

[2] On peut bien sûr écrire facilement un programme plus efficace, mais là n'est pas la question.

[3] Le qualificatif « persistant » est préférable à « purement fonctionnel » ou « immuable », qui sont trop réducteurs. Le sens exact de « persistant » est ici celui d'« observationnellement immuable », ainsi qu'expliqué dans le livre d'Okasaki [9].

réitérer, c'est itérer avec persistance

Il ne s'agit pas d'une technique originale. Les réitérateurs sont connus de certains programmeurs fonctionnels, et sont utilisés par exemple pour effectuer la comparaison deux à deux des éléments de deux arbres binaires de recherche (car il s'agit de s'arrêter dès qu'il y a différence) dans les bibliothèques standards d'OCAML ou de SML. Les solutions paraissent cependant à chaque fois relativement *adhoc*, et leur caractère persistant est souvent anecdotique (il n'est pas vraiment utile à cet endroit, mais juste là de fait car une programmation impérative n'est pas envisagée). Le but de cet article est une étude plus systématique des réitérateurs, dans différents cas de figure.

Cet article est décomposé en deux grandes parties. La première (section 2) décrit comment réaliser des réitérateurs pour plusieurs types de parcours sur les arbres binaires. La seconde (section 3) fait ensuite le lien avec le *zipper* de Gérard Huet [6]. L'intégralité du code OCAML correspondant à ce qui est décrit dans cet article est accessible en ligne à `http://www.lri.fr/~filliatr/pub/enum.ml`.

2. Réitérateurs pour les arbres binaires

Dans la suite de cet article, on suppose que la structure de données à parcourir est celle d'arbres binaires contenant des entiers aux nœuds :

```
type t = E | N of t × int × t
```

La généralisation à des arbres équilibrés — et donc contenant des informations supplémentaires aux nœuds — ou à des arbres contenant des éléments d'un autre type est immédiate car seuls les parcours nous intéressent ici.

Cette section décrit la réalisation de réitérateurs pour différents parcours des arbres binaires, en leur donnant à chaque fois la signature suivante :

```
type enum
val start : t → enum
val step : enum → int × enum
```

Comme indiqué dans l'introduction, il est convenu que la fonction `step` lève l'exception `Exit` lorsque l'énumération est terminée.

2.1. Parcours infixe

On commence par le parcours infixe, qui est le parcours le plus naturel lorsqu'il s'agit d'arbres binaires de recherche. Dans ce parcours, le sous-arbre gauche est examiné en premier, puis l'élément au nœud, puis le sous-arbre droit. Un itérateur d'ordre supérieur de type `iter` s'écrit donc ainsi :

```
let rec infixe f = function
  | E → ()
  | N (l, x, r) → infixe f l; f x; infixe f r
```

On trouve la réalisation d'un réitérateur correspondant à ce parcours dans la « littérature » (les bibliothèques standards d'OCAML et de SML par exemple). Puisque l'itération démarre sur l'élément le plus à gauche dans l'arbre, on commence par construire une fonction effectuant une descente vers la gauche et construisant la liste des éléments et sous-arbres droits rencontrés en chemin. On peut créer pour cela un type de listes spécifique :

```
type enum = End | More of int × t × enum
```

et une fonction `left` effectuant cette descente à gauche à partir d'un arbre `t` et d'une énumération `e` qui représente les éléments qui viendront après ceux de `t` :

```
let rec left t e = match t with
  | E → e
  | N (l, x, r) → left l (More (x, r, e))
```

On initialise alors l'énumération avec la « liste vide » `End` :

```
let start t = left t End
```

et la fonction `step` consiste alors à renvoyer l'élément en tête de liste et à appeler `left` avec ce qui était le sous-arbre droit de `x` dans l'arbre initial, soit `r`, puisque l'élément suivant `x` est justement le plus à gauche dans `r` :

```
let step = function
  | End → raise Exit
  | More (x, r, e) → x, left r e
```

2.2. Parcours préfixe

Le parcours préfixe consiste à examiner en premier l'élément au nœud, puis les éléments du sous-arbre gauche et enfin ceux du sous-arbre droit :

```
let rec préfixe f = function
  | E → ()
  | N (l, x, r) → f x; préfixe f l; préfixe f r
```

Cela correspond à un parcours en profondeur de l'arbre et c'est donc tout naturellement que le réitérateur peut être représenté par une *pile*, c'est-à-dire une liste d'arbres :

```
type enum = t list
```

L'énumération est initialisée avec la liste réduite à l'arbre de départ :

```
let start t = [t]
```

et la fonction `step` consiste à examiner l'arbre en tête de liste et, lorsqu'il s'agit d'un nœud, à renvoyer sa valeur en poussant sur la pile les sous-arbres droit et gauche (dans cet ordre) :

```
let rec step = function
  | [] → raise Exit
  | E :: e → step e
  | N (l, x, r) :: e → x, l :: r :: e
```

On peut même légèrement optimiser en évitant d'empiler des arbres vides :

```
let start = function E → [] | t → [t]
let step = function
  | [] → raise Exit
  | N (E, x, E) :: e → x, e
  | N (E, x, r) :: e → x, r :: e
  | N (l, x, E) :: e → x, l :: e
  | N (l, x, r) :: e → x, l :: r :: e
  | _ → assert false
```

Sur cet exemple, on constate que le réitérateur n'est rien d'autre que la concrétisation de la pile d'appels. Incidemment, cela démontre un autre avantage des réitérateurs : éviter le débordement de pile (*stack overflow* en anglais). Même si dans le cas d'arbres binaires équilibrés il est peu probable que la hauteur d'un arbre puisse être responsable d'un débordement de pile, le cas d'autres structures de données, tels que des graphes par exemple, peut être plus problématique pour des itérateurs écrits simplement comme des fonctions récursives sur la structure de données. Bien entendu, il est toujours possible d'expliciter la pile, même dans le cas des itérateurs d'ordre supérieur usuels.

2.3. Parcours postfixe

Le parcours postfixe consiste à examiner l'élément au nœud après avoir examiné les éléments de ses deux fils. De façon surprenante, le parcours postfixe est plus délicat à réaliser que le parcours préfixe. Bien sûr, on pourrait réutiliser l'idée d'expliciter la pile, et empiler aussi bien des arbres que des éléments (qu'on pourrait d'ailleurs représenter comme des arbres à un élément). Mais ce ne serait pas une solution très efficace. Plus judicieusement, on peut s'inspirer de la solution adoptée pour le parcours infixe, puisque dans le parcours postfixe l'élément le plus à gauche est également celui par lequel on commence. On reprend donc le même type de réitérateur et les mêmes fonctions `left` et `start` :

```
type enum = End | More of int × t × enum
let rec left t e = match t with
  | E → e
  | N (l, x, r) → left l (More (x, r, e))
let start t = left t End
```

Seule la fonction `step` change, qui doit faire maintenant passer le fils droit `r` *avant* l'élément `x` :

```
let rec step = function
  | End → raise Exit
  | More (x, E, e) → x, e
  | More (x, r, e) → step (left r (More (x, E, e)))
```

Le fait d'empiler l'arbre vide `E` avec `x` dans la dernière ligne n'est pas très élégant, et on peut raffiner cette solution en introduisant un constructeur particulier `More1` pour traiter ce cas particulier :

```
type enum = End | More of t × int × enum | More1 of int × enum
let rec left t e = match t with
  | E → e
  | N (l, x, E) → left l (More1 (x, e))
  | N (l, x, r) → left l (More (r, x, e))
let start t = left t End
let rec step = function
  | End → raise Exit
  | More1 (x, e) → x, e
  | More (t, x, e) → step (left t (More1 (x, e)))
```

2.4. Parcours en largeur d'abord

Pour terminer cette section sur les arbres binaires, considérons le cas du parcours en largeur d'abord. Un tel parcours est réalisé traditionnellement à l'aide d'une *file* contenant des arbres. Elle contient initialement l'arbre tout entier, et pour chaque nœud qui sort de la file on examine son élément et on insère dans la file ses deux fils (le gauche puis le droit). On peut ainsi écrire un itérateur d'ordre supérieur usuel à l'aide du module Queue d'OCAML aussi simplement que

```
let bfs f t =
  let q = Queue.create () in
  Queue.push t q;
  while not (Queue.is_empty q) do match Queue.pop q with
    | E → ()
    | N (l, x, r) → f x; Queue.push l q; Queue.push r q
  done
```

Pour réaliser le réitérateur correspondant, il suffit de substituer aux files impératives du module Queue des files *persistantes*. Il se trouve qu'il est très facile de coder de telles files à l'aide d'une paire de listes, tout en conservant de très bonnes performances [9]. Le code correspondant est donné en appendice, sous la forme d'un module Q fournissant un type abstrait α t pour des files contenant des éléments de type α. Le réitérateur est alors directement une file persistante contenant des arbres :

```
type enum = t Q.t
```

La fonction start construit la file ne contenant qu'un seul élément, à savoir l'arbre tout entier :

```
let start t = Q.push t Q.empty
```

et la fonction step applique exactement le même algorithme que le code impératif ci-dessus :

```
let rec step e =
  try match Q.pop e with
    | E, e → step e
    | N (l, x, r), e → x, Q.push r (Q.push l e)
  with Q.Empty →
    raise Exit
```

Note : de la même manière que nous l'avons fait pour le parcours préfixe, il est possible de légèrement optimiser en évitant d'insérer des arbres vides dans la file.

3. Liens avec le zipper

Dans cette seconde partie, nous étudions les relations qui existent entre les réitérateurs que nous venons de présenter et le *zipper* de Gérard Huet [6]. Plus précisément, nous allons montrer comment on peut retrouver ces réitérateurs en utilisant le *zipper* de manière systématique.

3.1. Le zipper

Nous présentons ici le *zipper* pour le lecteur qui n'en serait pas familier. Le *zipper* est à une structure de données purement applicative ce que le pointeur est à une structure de données impérative : un moyen d'en désigner une portion et de la modifier. Dans le cas d'une structure purement applicative, « modifier » signifie évidemment construire une nouvelle valeur, mais le problème n'en reste pas moins difficile. Imaginons par exemple qu'on souhaite parcourir les nœuds d'un arbre binaire à la recherche d'un nœud vérifiant une certaine propriété et, une fois ce nœud trouvé, d'y effectuer une modification locale. Avec une structure impérative, cela ne pose aucun problème. Mais avec une structure purement applicative, il faut maintenir en permanence le chemin depuis la racine de l'arbre jusqu'au nœud en cours d'examen, de manière à pouvoir reconstruire les nœuds correspondants. C'est exactement ce que fait le *zipper*, avec la plus grande élégance.

Un tel chemin depuis la racine est représenté à l'envers, comme une liste allant du nœud examiné vers la racine de l'arbre, la direction suivie étant indiquée à chaque étape. Le type OCAML représentant ce chemin est le suivant :

```
type path = Top | Left of path × int × t | Right of t × int × path
```

Le *zipper* proprement dit est alors un couple formé du sous-arbre « pointé » et de son chemin jusqu'à la racine :

```
type location = t × path
```

Cette construction peut être généralisée à n'importe quel type algébrique, chaque constructeur étant dupliqué en autant de variantes qu'il possède de sous-termes du type en question (ici c'est le constructeur `N` qui est dupliqué en `Left` et `Right`).

On crée un *zipper* pointant sur la racine d'un arbre `t` en l'associant au chemin vide :

```
let create t = (t, Top)
```

et on peut ensuite construire des fonctions de *navigation* permettant de se déplacer dans l'arbre représenté par le *zipper*. Ainsi, pour descendre vers le fils gauche, lorsqu'il existe, il suffit d'allonger le chemin avec le constructeur `Left` en y conservant la valeur du nœud et le fils droit, et de prendre le fils gauche comme nouvel arbre pointé :

```
let go_down_left = function
  | E, _ → invalid_arg "go_down_left"
  | N (l, x, r), p → l, Left (p, x, r)
```

De façon symétrique, on peut écrire une fonction pour descendre vers le fils droit :

```
let go_down_right = function
  | E, _ → invalid_arg "go_down_right"
  | N (l, x, r), p → r, Right (l, x, p)
```

De même, on peut écrire des fonctions pour passer d'un sous-arbre à son frère gauche ou à son frère droit, lorsqu'ils existent :

```
let go_left = function
  | _, Top | _, Left _ → invalid_arg "go_left"
  | r, Right (l, x, p) → l, Left (p, x, r)

let go_right = function
  | _, Top | _, Right _ → invalid_arg "go_right"
  | l, Left (p, x, r) → r, Right (l, x, p)
```

Enfin, on peut écrire une fonction permettant de remonter d'un nœud dans l'arbre :

```
let go_up = function
  | _, Top → invalid_arg "go_up"
  | l, Left (p, x, r) | r, Right (l, x, p) → N (l, x, r), p
```

Itérer cette fonction jusqu'à obtenir le chemin vide `Top` est un moyen de retrouver l'arbre entier représenté par le *zipper*. La modification locale proprement dite, motivation initiale du *zipper*, est trivialement effectuée en remplaçant le sous-arbre pointé par le *zipper* par un nouvel arbre :

```
let change (_, p) t = (t, p)
```

On note que toutes ces opérations sont réalisées en temps et espace constant.

3.2. Réitérateurs dérivés du zipper

Nous montrons maintenant comment le *zipper* permet de retrouver les réitérateurs sur les arbres binaires présentés à la section 2.

3.2.1. Parcours infixe

Commençons par le parcours infixe. Le réitérateur est directement représenté par un *zipper* et la fonction `start` place le *zipper* à la racine de l'arbre :

```
type enum = location
let start t = (t, Top)
```

La fonction `step` est alors réalisée en combinant les fonctions de navigation offerte par le *zipper*, ainsi que celle de modification locale pour faire disparaître un à un les éléments énumérés. Si l'arbre tout entier est vide, alors l'énumération est terminée :

```
let rec step = function
  | E, Top → raise Exit
```

Si le sous-arbre examiné est un nœud, alors il convient de continuer à descendre vers la gauche, avec la primitive `go_down_left` du *zipper*, et de rappeler `step`. Si on expanse `go_down_left`, on obtient :

```
  | N (l, x, r), p → step (l, Left (p, x, r))
```

Enfin, si on atteint l'arbre vide `E`, alors l'élément `x` juste au dessus est celui qu'il convient de renvoyer, et on peut alors remplacer le nœud en question par son fils droit. Cela s'obtient par une combinaison des primitives `go_up` et `change` qui une fois expansées donne le code suivant :

```
  | E, Left (p, x, r) → x, (r, p)
```

Au final, la fonction `step` s'écrit donc aussi facilement que

```
let rec step = function
  | E, Top → raise Exit
  | E, Left (p, x, r) → x, (r, p)
  | N (l, x, r), p → step (l, Left (p, x, r))
```

On constate que le constructeur `Right` du *zipper* n'a pas servi et on peut donc le supprimer :

```
type path = Top | Left of path × int × t
type enum = t × path
```

On retrouve alors *exactement* le type qui avait été introduit section 2, à savoir une liste de couples formés d'éléments et de leurs fils droits associés. La fonction `left` de la solution initiale a disparu : elle est maintenant réalisée directement par la fonction `step`. Le comportement est cependant légèrement différent : certains appels à `left` dans la solution initiale sont ici suspendus dans la première composante du *zipper*, et ne seront effectués qu'au coup suivant. Si l'on stoppe une itération sur un élément dont le fils droit est énorme (à gauche), alors on économise la descente vers la gauche dans ce sous-arbre. La solution inspirée par le *zipper* est donc légèrement plus efficace.

3.2.2. Parcours préfixe

On réalise ici encore le réitérateur directement par le *zipper* et la fonction `start` ne change pas, tout comme le cas de terminaison de l'énumération :

```
let rec step = function
  | E, Top → raise Exit
```

Si le sous-arbre examiné est un nœud, on retourne directement l'élément correspondant, et on déplace le *zipper* vers le fils gauche (`go_down_left`) :

```
  | N (l, x, r), p → x, (l, Left (p, x, r))
```

Enfin, lorsque l'itération atteint l'élément le plus à gauche dans l'arbre, alors on saute directement au sous-arbre droit correspondant car l'élément au nœud a déjà été visité :

```
  | E, Left (p, _, r) → step (r, p)
```

Au final, on a donc le code suivant :

```
let rec step = function
  | E, Top → raise Exit
  | E, Left (p, _, r) → step (r, p)
  | N (l, x, r), p → x, (l, Left (p, x, r))
```

Là encore, on note que le constructeur `Right` n'est pas utilisé, tout comme l'élément contenu dans le constructeur `Left`. On peut donc simplifier la définition du réitérateur en

```
type path = Top | Left of path × t
type enum = t × path
```

Une fois encore, on constate qu'on retrouve *exactement* le même type que dans la solution initiale, à savoir une liste d'arbres (avec un *cons* particulier pour le premier élément de la liste, sous forme d'une paire). En terme d'efficacité, cette solution inspirée par le *zipper* se situe entre les deux solutions proposées section 2, car elle évite l'empilement de certains arbres vides, mais pas tous.

3.2.3. Parcours postfixe

On réalise toujours le réitérateur par le *zipper* et on ne change toujours pas la fonction `start` et le cas de terminaison de `step` :

```
let start t = (t, Top)
let rec step = function
  | E, Top → raise Exit
```

Si le sous-arbre examiné est un nœud, il convient de descendre dans le fils gauche :

```
  | N (l, x, r), p → step (l, Left (p, x, r))
```

Si l'itération en a terminé avec un sous-arbre gauche, elle doit considérer maintenant le sous-arbre droit (son frère) :

```
  | E, Left (p, x, r) → step (r, Right (E, x, p))
```

Enfin, si l'itération atteint l'élément le plus à droite, il convient de le renvoyer et de simplement supprimer le nœud correspondant :

```
  | E, Right (_, x, p) → x, (E, p)
```

Au final, on obtient le code suivant :

```
let rec step = function
  | E, Top → raise Exit
  | E, Left (p, x, r) → step (r, Right (E, x, p))
  | E, Right (_, x, p) → x, (E, p)
  | N (l, x, r), p → step (l, Left (p, x, r))
```

Ici, les deux constructeurs du *zipper* sont utilisés, mais on note que le premier argument du constructeur `Right` reste inutilisé, ce qui permet de simplifier légèrement la définition du réitérateur en

```
type path = Top | Left of path × int × t | Right of int × path
type enum = t × path
```

et de retrouver une fois encore un type isomorphe à celui introduit section 2 (`Left` correspondant à `More` et `Right` à `More1`).

3.2.4. Parcours en largeur d'abord

Le cas du parcours en largeur d'abord est plus complexe. En effet, utiliser les seules primitives de navigation offertes par le *zipper* pour passer d'un nœud au nœud qui le suit dans le parcours en largeur d'abord se révèle très difficile : il faut remonter plus ou moins haut dans l'arbre et redescendre selon un chemin bien spécifique, d'une manière qui dépend de la structure *globale* de l'arbre.

Comme bien souvent avec les problèmes de parcours en largeur, il convient de généraliser le problème à des *forêts* (voir par exemple [10]), c'est-à-dire à des listes d'arbres. En effet, on pourra alors représenter la forêt constituée des sous-arbres de même niveau, et se déplacer directement d'un nœud à celui qui se trouve à sa droite dans cette forêt.

Il se trouve qu'il existe un *zipper* pour les arbres d'arité variable, donc pour les forêts. Dans l'article introduisant le *zipper* [6], le cas des arbres d'arité variable est même présenté avant le cas particulier des arbres binaires. Le *zipper* est ainsi défini :

```
type path = Top | Node of t list × path × t list
type location = t × path
```

Les trois arguments du constructeur `Node` représentent une position dans une forêt, la première liste donnant les arbres situés à gauche *en ordre inverse* et la seconde liste les arbres situés à droite. Les primitives de navigation qui nous intéressent ici sont alors les suivantes :

```
let go_left = function
  | t, Node (l :: ll, p, r) → l, Node (ll, p, t :: r)
  | _ → invalid_arg "go_left"

let go_right = function
  | t, Node (l, p, r :: rr) → r, Node (t :: l, p, rr)
  | _ → invalid_arg "go_right"
```

Comme dans les autres cas, le réitérateur est réalisé directement par le *zipper* et la fonction `start` place le *zipper* sur la racine de l'arbre :

```
type enum = location
let start t = t, Node ([], Top, [])
```

Comme précédemment, la fonction `step` est réalisée en se déplaçant grâce aux fonctions de navigation et en supprimant les nœuds au fur et à mesure. Le cas de la forêt vide termine l'énumération :

```
let rec step = function
  | E, Node ([], p, []) → raise Exit
```

Si le sous-arbre examiné est un nœud, on retourne directement l'élément correspondant et on remplace le nœud par ses deux fils, empilés à gauche dans la forêt. Et pour éviter de traiter trop de cas particuliers, on remplace le sous-arbre pointé par un arbre vide :

```
  | N (l, x, r), Node (ll, p, rr) → x, (E, Node (r :: l :: ll, p, rr))
```

Si le sous-arbre examiné est justement un arbre vide, alors on se déplace vers la droite dans la forêt :

```
  | E, Node (ll, p, r :: rr) → step (r, Node (ll, p, rr))
```

Enfin, s'il n'est plus possible de se déplacer vers la droite, il convient de revenir tout à gauche dans la forêt (pour passer au niveau suivant). Cela revient à appliquer la fonction `go_left` autant de fois que possible, et aura donc pour effet de retourner la liste d'arbres de gauche dans la liste droite (de la manière efficace, c'est-à-dire en utilisant un accumulateur, en l'occurrence la liste d'arbres de droite). On peut donc utiliser directement `List.rev` :

```
  | E, Node (ll, p, []) → step (E, Node ([], p, List.rev ll))
```

Au final on obtient donc le code suivant :

```
let rec step = function
  | E, Node ([], p, []) → raise Exit
  | E, Node (ll, p, []) → step (E, Node ([], p, List.rev ll))
  | E, Node (ll, p, r :: rr) → step (r, Node (ll, p, rr))
  | N (l, x, r), Node (ll, p, rr) → x, (E, Node (r :: l :: ll, p, rr))
  | _, Top → assert false
```

Mais on constate immédiatement que le *zipper* est toujours de la forme `Node(_,Top,_)`. On peut donc supprimer les constructeurs `Top` et `Node` et représenter directement le *zipper* par une paire de listes. On peut également mettre le sous-arbre pointé en tête de la seconde liste, ce qui nous donne au final le code suivant :

```
type enum = t list × t list
let start t = [], [t]
let rec step = function
  | [], [] → raise Exit
  | ll, [] → step ([], List.rev ll)
  | ll, E :: rr → step (ll, rr)
  | ll, N (l, x, r) :: rr → x, (r :: l :: ll, rr)
```

C'est *très exactement* la solution donnée section 2.4, si ce n'est que le codage des files persistantes par des paires de listes (voir l'appendice) est ici directement écrit dans la fonction `step`. Incidemment, nous avons retrouvé le codage efficace des files persistantes en cherchant à utiliser le *zipper* pour effectuer un parcours en largeur.

parcours	itérateur	aléatoires	gauche	droite	complets
infixe	section 2.1	1.07	0.86	0.11	0.25
	zipper	1.14	1.12	0.11	0.27
	cps	1.28	1.38	0.12	0.29
préfixe	section 2.2	1.01	0.76	0.10	0.26
	— variante	0.91	0.08	0.07	0.21
	zipper	0.99	0.99	0.11	0.27
	cps	1.51	0.14	0.16	0.41
postfixe	section 2.3	1.26	0.86	1.04	0.28
	— variante	1.20	0.69	0.87	0.29
	zipper	1.42	1.08	1.06	0.35
	cps	1.63	1.44	1.21	0.43
largeur	section 2.4	14.57	0.44	0.47	4.62
	— variante	9.26	0.24	0.24	2.05
	zipper	15.38	0.18	0.17	3.65

FIG. 1 – Performances comparées des différentes solutions

4. Performances

Dans cette section, nous comparons rapidement l'efficacité des diverses solutions proposées dans cet article. La figure 1 rassemble les mesures effectuées pour les différents types de parcours et les différentes solutions. Pour chaque itérateur persistant, on mesure le temps de parcours d'une centaine d'arbres comprenant de 0 à 100 000 éléments pour quatre types d'arbres différents : arbres aléatoires, arbres linéaires à gauche, arbres linéaires à droite et enfin arbres complets. Les parcours faisant référence aux solutions données dans la section 2 sont nommés « section 2.1 » à « section 2.4 » et « variante » fait référence aux variantes consistant à ne jamais empiler d'arbre vide ; « zipper » correspond aux solutions présentées dans la section 3 ; enfin, « cps » correspond à une dernière solution, non présentée dans cet article, consistant à représenter l'itérateur persistant directement par une fonction, c'est-à-dire où les clôtures d'OCAML sont utilisées comme structure de données[4] :

```
type enum = unit → int × enum
```

Le codage correspondant à cette solution est disponible en ligne avec les autres solutions présentées dans cet article (`http://www.lri.fr/~filliatr/pub/enum.ml`).

On constate que les itérateurs décrits section 2 sont très légèrement plus efficaces que les versions dérivées du *zipper*, ce qui s'explique par le fait que la structure de données est légèrement plus immédiate (il n'y a pas la paire que l'on trouve au sommet du *zipper*), mais la différence n'est pas vraiment significative. Dans le cas du parcours en largeur d'abord, on observe que la solution inspirée du *zipper* est parfois plus efficace, mais cela s'explique par le fait qu'elle correspond à un dépliage du module `Q` réalisant des files persistantes. Enfin, on observe que la solution « cps » est toujours moins efficace, ce qui s'explique par une représentation plus gourmande : les clôtures coûtent ici plus cher que les types de données *adhoc*.

Nous avons également effectué une comparaison similaire des performances en mémoire, en mesurant l'évolution au fur et à mesure des parcours de la quantité totale de mémoire utilisée par l'itérateur persistant. Les résultats sont comparables à ceux des performances en temps : les solutions de la section 2 et celles inspirées du *zipper* utilisent des quantités quasi identiques de mémoire, là où la solution « cps » en utilise beaucoup plus (mais dans un rapport constant).

[4]Une telle définition nécessite l'option `-rectypes` du compilateur OCAML.

Les résultats de tous ces tests de performance peuvent être reproduits à partir du code source mis en ligne.

5. Conclusion

Dans cet article, nous avons présenté une alternative aux itérateurs traditionnels qu'on trouve en ML, sous la forme d'itérateurs pas à pas reposant sur des structures persistantes. Ces *réitérateurs* permettent notamment l'interruption prématurée d'une itération et mieux encore la reprise d'une itération sur un état antérieur lorsqu'ils sont impliqués dans un algorithme effectuant du *backtracking*.

De tels itérateurs sont connus de certains programmeurs, quoique leur caractère persistant n'est en fait pas exploité. Ils restent de toutes façons peu répandus et mériteraient plus de publicité. Nous avions déjà fait un pas dans ce sens en fournissant dans la bibliothèque OCAMLGRAPH [2, 4] des réitérateurs pour les parcours de graphes en profondeur et en largeur. On pourrait poursuivre dans cette voie en proposant des réitérateurs pour d'autres structures de données usuelles (tables de hachage, files, piles, etc.).

Cet article a également démontré les liens étroits qui existent entre les réitérateurs et le *zipper* de Gérard Huet, et notamment comment nous avons pu (re)trouver facilement les réitérateurs correspondant à divers parcours sur les arbres binaires à partir des primitives offertes par le *zipper*. Pour compléter cette analyse, il conviendrait d'étudier également les solutions offertes par la construction `call/cc` en matière d'itération pas à pas — même si cette construction n'est pas disponible en OCAML — et de compléter ainsi le diagramme des relations existant entre les réitérateurs, le *zipper* et `call/cc`.

Remerciements. Je remercie chaleureusement Sylvain Conchon, Benjamin Monate et Julien Signoles pour les discussions informelles que j'ai pu avoir avec eux sur ce sujet et pour leurs commentaires sur cet article. Je remercie le rapporteur pour ses suggestions.

Références

[1] Le langage Objective Caml. `http://caml.inria.fr/`.

[2] Ocamlgraph, une bibliothèque de graphes pour Objective Caml. `http://www.lri.fr/~filliatr/ocamlgraph/`.

[3] Dariusz Biernacki, Olivier Danvy, and Chung chieh Shan. On the Static and Dynamic Extents of Delimited Continuations. Technical Report RS-05-36, BRICS, 2005. To appear in the journal Science of Computer Programming.

[4] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Le foncteur sonne toujours deux fois. In *Seizièmes Journées Francophones des Langages Applicatifs*. INRIA, Mars 2005.

[5] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[6] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5) :549–554, Septembre 1997.

[7] Oleg Kiselyov. From enumerators to cursors : turning the left fold inside out, 2004. At `http://okmij.org/ftp/Haskell/fold-stream.lhs`.

[8] John McCarthy. Another samefringe. *SIGART Newsletter*, 61, February 1977.

[9] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[10] Chris Okasaki. Breadth-First Numbering : Lessons from a Small Exercise in Algorithm Design. In *International Conference on Functional Programming (ICFP)*, Montreal, Canada, 2000.

[11] Richard Rasala. A Model C++ Tree Iterator Class For Binary Search Trees. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education SIGCSE*, 1997.

Appendice : files persistantes

Nous donnons ici une réalisation simple mais efficace des files persistantes [9]. L'idée est de représenter une file par une paire de listes, l'une pour y mettre les éléments (en tête) et l'autre pour les en retirer (toujours en tête). On est éventuellement amené à retourner la première liste lorsque la seconde est épuisée, mais la complexité des opérations `push` et `pop` reste $O(1)$ en temps amorti i.e. ramené à la totalité des opérations.

```
module Q : sig
  type α t
  exception Empty
  val empty : α t
  val is_empty : α t → bool
  val push : α → α t → α t
  val pop : α t → α × α t
end = struct
  type α t = α list × α list
  exception Empty
  let empty = [], []
  let is_empty = function [], [] → true | _ → false
  let push x (i,o) = (x :: i, o)
  let pop = function
    | i, y :: o → y, (i,o)
    | [], [] → raise Empty
    | i, [] → match List.rev i with
        | x :: o → x, ([], o)
        | [] → assert false
  let peek q = fst (pop q)
end
```