Programmation Fonctionnelle Avancée
Les zippers

Ralf Treinen

Université Paris Diderot
UFR Informatique
Institut de Recherche en Informatique Fondamentale

PARIS DIDEROT université

treinen@irif.fr

24 septembre 2018

Description des objectifs

Naviguer dans une structure de données, par exemple une liste (en avant, et en arrière) :

- en gardant la possibilité d'ajouter des valeurs au milieu
- sans faire de copies
- sans pointeurs arrières
- *et de façon efficace*

Des fonctions sur les listes avec position

- Premier essai : modéliser des listes avec une position :

  $$(\text{position\_actuelle}, \text{dernière\_position}, \text{liste})$$

- Avantage : mouvement à gauche/droite en temps linéaire.
- Inconvénient : il faut effectivement parcourir la liste du début jusqu'à la position actuelle pour insérer/supprimer.

Exemples (list1.ml)

```
(* une liste avec une position et une longueur *)
type 'a listpos = int * int * 'a list;;
(* changer de position, temps constant *)
let agauche = function
  | (0,_,_) -> failwith "debut de liste"
  | (p,t,l) -> (p-1,t,l);;
let adroite = function
  | (p,t,l) when p=t -> failwith "fin de liste"
  | (p,t,l) -> (p+1,t,l)
;;
let from_list l = (0, (List.length l)-1, l);;
let get_current (p,t,l) = List.nth l p;;
let x = from_list ['a'; 'b'; 'd'];;
get_current(adroite (adroite x));;
```

Exemples (list2.ml)

```
(* insertion : temps lineaire *)
let insert_at_point x (p,t,l) =
   let rec ins =
      function
      | (0,l) -> x::l
      | (n,y::r) -> y::(ins (n-1,r))
      | _ -> assert false
   in (p,t+1,ins(p,l));;

  x;;
let y = insert_at_point 'c' (adroite (adroite x));;
```


Exemples (list3.ml)

```
(* suppression : temps lineaire *)
let delete_at_point (p,t,l) =
   let rec del =
      function
      | (0,x::l) -> l
      | (n,y::r) -> y::(del (n-1,r))
      | _ -> assert false
   in if p=t then failwith "fin de liste"
             else (p,t-1,del(p,l))
;;

let z = delete_at_point y;;
```


Des fonctions sur les listes avec position

Ce n'est pas ce qu'on veut

- programmation pénible avec des compteurs ;
- on ne voit pas, dans le toplevel, ou l'on est dans la structure de données ;
- ce n'est pas efficace : l'insertion et la suppression prennent un temps non constant.

Listes doublement chaînées

- On peut essayer avec des références.
- On utilise des enregistrements avec le contenu d'une cellule, et des références vers les cellules précédente et suivante.

```
(* careful, danger ahead ! *)
type 'a listdl = Nil | Cell of 'a cell
and 'a cell = {info:'a;
               next:'a listdl ref;
               prev:'a listdl ref
              };;
```



```
(* insertion, temps constant *)
let insert a = function
  | Nil -> Cell {info=a;
                 prev=ref Nil;next=ref Nil}
  | Cell c as dll ->
     let p = !(c.prev) in
     let c' = Cell {info=a;
                    prev=ref p;
                    next=ref dll} in
    c.prev:=c';
    begin
      match p with
      | Nil -> ()
      | Cell cp -> cp.next:=c'
    end;
    c'
;;
let l = insert 1 Nil ;;
let l'= insert 2 l ;; (* cycles ! *)
```



```
let rec of_list = function
  | [] -> Nil
  | a::r -> let dll = of_list r in
            let c = Cell {info=a;next=ref dll;prev = ref Nil} in
            match dll with
              Nil -> c
            | Cell c' -> c'.prev := c; c
;;
let rec toutagauche = function
  | Nil -> Nil
  | Cell {prev=c} as dll when !c = Nil -> dll
  | Cell {prev=c} -> toutagauche !c
;;
let to_list dll = let dll' = toutagauche dll in
                  let rec aux acc = function
                    | Nil -> acc
                    | Cell {info=a;next=c} -> aux (a::acc) !c
                  in List.rev (aux [] dll')
;;
```


Exemples (list7.ml)

```
(* deplacement, temps constant *)
let agauche = function
  | Nil -> failwith "Liste vide"
  | Cell {prev=c} when !c = Nil -> failwith "Deja a gauc
  | Cell {prev=c} -> !c
;;

let adroite = function
  | Nil -> failwith "Liste vide"
  | Cell {next=c} when !c = Nil -> failwith "Deja a droi
  | Cell {next=c} -> !c
;;

let x = of_list ['a'; 'b'; 'd'] ;;
let y = to_list(insert 'c' (adroite (adroite x)));;
```

Listes doublement chaînées

Ce n'est pas idéal

- fonctions d'impression du toplevel inutiles (les pointeurs arrière font des cycles)
- programmation pénible et avec des effets de bord (la structure de données n'est plus persistante)
- il nous faut une idée !

Zipper de listes

- Structure de données popularisée par Gérard Huet 1997.
- Idée : représenter une liste comme deux piles, une pour les éléments à gauche de la position actuelle dans la liste, une pour les éléments à droite.
- Les éléments sur les sommets des deux piles sont les voisins de la position actuelle dans la liste.
- Zipper = Fermeture éclaire

Exemples (ziplist1.ml)

```
(* un bloc sur la pile contient juste un 'a *)
type 'a pile = 'a list
type 'a listzipper = 'a pile * 'a list

exception Zipper of string;;

(* navigation en temps lineaire *)
let agauche : 'a listzipper -> 'a listzipper = function
  | ([],_)   -> raise (Zipper "Deja a gauche")
  | (a::p,l) -> (p, a::l);;
let adroite : 'a listzipper -> 'a listzipper  = function
  | (p,[])   -> raise (Zipper "Deja a droite")
  | (p,a::l) -> (a::p, l);;
```


Exemples (ziplist2.ml)

```
(* conversion *)
let from_list l = ([], l)
;;

let to_list (c, l) =
  let rec revapp c l = match c with
    | [] -> l
    | h::r -> revapp r (h::l)
  in revapp c l
;;

to_list (['a'; 'b'; 'c'], ['d'; 'e'; 'f']);;
```

Exemples (ziplist3.ml)

```
(* insert et delete en temps lineaire *)
let insert v = function
    (pile,liste) -> (pile, v::liste);;

let delete = function
| (p,[])    -> raise (Zipper "Trop a droite pour effacer")
| (p,a::r) -> (p, r)
;;

  to_list(insert 'c' (adroite(adroite(from_list
                                        ['a'; 'b'; 'd'; 'e'
```


L'intuition derrière les zippers

Considérons une visite récursive d'une liste

```
let rec visit = function
|[] -> ()
| a::r -> visit r
```

Lors de la visite de `[1;2;3;4]`, la valeur conservée sur la pile d'appel, et la valeur du parametre évoluent comme suit :

`[]`	`[1;2;3;4]`
`[1]`	`[2;3;4]`
`[2;1]`	`[3;4]`
`[3;2;1]`	`[4]`
`[4;3;2;1]`	`[]`

L'intuition derrière les zippers, bis

En général

Un zipper représente *explicitement* l'état de la pile d'appel, et la valeur courante, lors d'une visite récursive d'une structure de données.

- La pile d'appel est une suite de *blocs* d'activation.
- Chaque bloc contient les éléments de la structure de données qui ne sont pas passés à l'appel récursif, plus *un marqueur* indiquant sur quel sous-structure on continue la visite.

Dans le cas des listes, on n'a pas utilisé de marqueur, parce que il y a une seule soustructure pour continuer la visite.

Autre intuition derrière les zippers

- Une paire consistent en la sous-structure qu'on a selectionnée, et le *contexte* de la sous-structure.
- Le contexte est representée à l'envers (“inside-out”), c'est l'ordre naturel pour reconstruire la structure complète à partir de la sous-structure.

Zipper des arbres binaires

- arbres binaires, avec données sur les nœuds :

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

- un contexte est une suite de blocs, où
- un bloc consiste en
  1. un marqueur qui indique si la position se trouve à gauche ou à droite
  2. une donnée (à mettre sur un nœud)
  3. un arbre

Exemples (zipbintree1.ml)

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

type marqueur = Gauche | Droite

type 'a block = marqueur * 'a * 'a arbre

type 'a pile = 'a block list

type 'a arbrezipper = 'a pile * 'a arbre;;

exception Zipper of string;;
```


Exemples (zipbintree2.ml)

```
let bas_a_gauche : 'a arbrezipper -> 'a arbrezipper =
function (pile, arbre) -> match arbre with
| Feuille -> raise (Zipper "Feuille")
| Noeud (x, t1, t2) -> (Gauche, x, t2)::pile, t1;;

let bas_a_droite : 'a arbrezipper -> 'a arbrezipper =
function (pile, arbre) -> match arbre with
| Feuille -> raise (Zipper "Feuille")
| Noeud (x, t1, t2) -> (Droite, x, t1)::pile, t2;;

let en_haut: 'a arbrezipper -> 'a arbrezipper =
function (pile, arbre) -> match pile with
| (Gauche, x, t)::p -> p, Noeud (x, arbre, t)
| (Droite, x, t)::p -> p, Noeud (x, t, arbre)
| [] -> raise (Zipper "Racine");;
```


Zipper et arbres *n*-aires

- Pas de données sur le nœuds, mais chaque nœud a une *liste* de fils :

```
type 'a narbre =
  | Feuille of 'a
  | Noeud of 'a narbre list;;
```

- On veut pouvoir naviguer d'un nœud vers son père, son fils le plus à gauche, et à ses frères à gauche et à droite.
- Pour naviguer dans la liste des frères on utilise la structure des zippers pour des listes que nous connaissons déjà.

Exemples (ziptrees1.ml)

```
type 'a narbre =
  | Feuille of 'a
  | Noeud of 'a narbre list;;

type 'a listzipper = 'a list * 'a list

type 'a block = ('a narbre) listzipper

type 'a pile = 'a block list

type 'a narbrezipper = 'a pile * 'a narbre;;
```


Exemples (ziptrees2.ml)

```
let a_gauche : 'a narbrezipper -> 'a narbrezipper =
function (pile, arbre) -> match pile with
| (a::lp,l)::p -> (lp,arbre::l)::p,a
| ([],_)::p -> raise (Zipper "Deja a gauche")
| _ -> failwith "Racine";;

let a_droite : 'a narbrezipper -> 'a narbrezipper =
function (pile, arbre) -> match pile with
| (lp,a::l)::p -> (arbre::lp,l)::p,a
| (_,[])::pile' -> raise (Zipper "Deja a droite")
| _ -> raise (Zipper "Racine");;
```

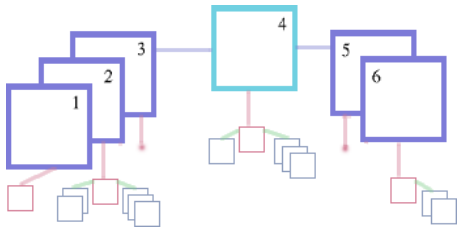

Exemples (ziptrees3.ml)

```
let en_bas : 'a narbrezipper -> 'a narbrezipper =
function (pile, arbre) -> match arbre with
| Noeud (first_son::older_sons) -> ([],older_sons)::pile, firs
| Noeud _ | Feuille _ -> raise (Zipper "Deja en bas");;

let en_haut : 'a narbrezipper -> 'a narbrezipper =
function (pile, arbre) -> match pile with
| (lp,l)::p -> p,Noeud(List.rev(lp)@(arbre::l))
| _ -> raise (Zipper "Deja en haut");;
```


Utilisations

- Huet était motivé par un éditeur structuré de preuves
- Librairie Clojure `http://clojure.org/other_libraries`
- Le gestionnaire de fenêtres XMonad (Haskell)



In this picture we have a window manager managing 6 virtual workspaces. Workspace 4 is currently on screen. Workspaces 1, 2, 4 and 6 are non-empty, and have some windows. The window currently receiving keyboard focus is window 2 on the workspace 4. The focused windows on the other non-empty workspaces are being tracked though, so when we view another workspace, we will know which window to set focus on. Workspaces 3 and 5 are empty.

Peut-on construire automatiquement des zippers pour un type quelconque ?

Dériver automatiquement des zippers pour un type quelconque serait un vrai plus pour un programmeur...
En 2001, Conor McBride observe un phénomène intéressant qui permet de faire ça en analogie avec la *dérivation de fonctions*.

Les étapes fondamentales sont les suivantes :

- on traduit (une sousclasse) des types d'OCaml dans le langage plus simple des *types récursifs*
- on définit formellement la *dérivée* d'un type récursif
- on obtient la définition du type du bloc de pile à partir de cette dérivée

Les types recursifs polynomiaux

On appelle *types récursifs polynomiaux* les types produits par la grammaire suivante, ou x est une variable prise dans un ensemble denombrable

$$F ::= x \mid 0 \mid 1 \mid F + F \mid F \times F \mid \mu x.F$$

Une definition de type en OCaml (ou Haskell, ou SML/NJ, ...) *sans types fonctionnels* peut se traduire naturellement dans ces types.

```
type 'a list = Nil | Cons of 'a * 'a list
```

devient, en oubliant les noms des constructeurs, et en rendant explicite la définition récursive

$$\mu x.1 + \text{'a} \times x$$

Quelques autres exemples

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

devient, en oubliant les noms des constructeurs, et en rendant explicite la définition récursive

$$\mu x.1 + \text{'a} \times x \times x$$

```
type 'a ntree = Feuille of 'a | Node of 'a ntree list
```

devient, en oubliant les noms des constructeurs, et en rendant explicite la définition récursive

$$\mu x.\text{'a} + x\ \mathit{list}$$

c'est à dire

$$\mu x.\text{'a} + (\mu y.1 + x \times y)$$

Rappel sur les dérivées des fonctions

On se rappelle, du cours d'Analyse, les formules suivantes :

$$\begin{aligned}
\partial_x x &= 1 \\
\partial_x y &= 0\ (x \neq y) \\
\partial_x a &= 0\ (a\ \textit{constante} : 1, 0, \text{'a}\ \textit{etc.}) \\
\partial_x (A + B) &= \partial_x A + \partial_x B \\
\partial_x (A \times B) &= \partial_x A \times B + A \times \partial_x B
\end{aligned}$$

On les applique, sans états d'âme, à nos définitions de type.

Le bloc de pile d'un zipper pour T est la dérivée de T

McBride observe qu'on trouve naturellement *le type du bloc de pile* pour le zipper d'un type polynomial en procédant comme suit :

- on construit le type récursif $\mu x.F$ associé

  par exemple, pour les listes, on obtient $\mu x.1 + \texttt{'a} \times x$ et $F = 1 + \texttt{'a} \times x$
- on calcule la dérivée formelle $\partial_x F$ de F par rapport à x

  pour les listes, on a

  $\partial_x(1+\texttt{'a}\times x) = \partial_x 1+\partial_x(\texttt{'a}\times x) = 0+\texttt{'a}\times\partial_x x = \texttt{'a}\times 1 = \texttt{'a}$
- on remplace dans $\partial_x F$ toute occurrence de x par $\mu x.F$
  pour l'exemple des listes, cela ne change rien.

En vérifiant notre code pour les zippers des listes, on voit que le *type du bloc de pile* est bien `'a` !

Pour en savoir plus

Conor McBride.
The derivative of a regular type is its type of one-hole contexts.
2001.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride.
Derivatives of containers.
In Hofmann [Hof03], pages 16–30.

Martin Hofmann, editor.
Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings, volume 2701 of Lecture Notes in Computer Science. Springer, 2003.

Les arbres binaires

Vérifions sur les arbres binaires :

- on construit le type récursif $\mu x.F$ pour les arbres binaires :
  $\mu x.1 + \texttt{'a} \times x \times x$, avec $F = 1 + \texttt{'a} \times x \times x$
- on calcule la dérivée formelle $\partial_x F$ de F par rapport à x
  la partie intéressante est

  $\partial_x(\texttt{'a} \times x \times x) = \partial_x \texttt{'a} \times x \times x + \texttt{'a} \times \partial_x(x \times x) = \texttt{'a} \times (x + x)$
- on remplace dans $\partial_x F$ toute occurrence de x par $\mu x.F$
  on obtient pour les arbres
  $\texttt{'a} \times ((\mu x.1 + \texttt{'a} \times x \times x) + (\mu x.1 + \texttt{'a} \times x \times x))$
  ce qui revient à $\texttt{'a} \times (\texttt{'a tree} + \texttt{'a tree})$

Dans notre code pour les zippers des arbres, le *type du bloc de pile* est bien constitué d'un couple contenant un `'a` et soit un arbre pris à gauche, soit un arbre pris à droite (cela correspond au type $\texttt{'a tree} + \texttt{'a tree}$)