Programmation Fonctionnelle Avancée
Les zippers

Pierre Letouzey

Université de Paris
UFR Informatique
Institut de Recherche en Informatique Fondamentale
letouzey@irif.fr

9 février 2022

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 en la position p *)
let insert_at_point x (p,t,l) =
    let rec ins n l =
      if n=0 then x::l
      else match l with
        | [] -> assert false
        | y::r -> y::(ins (n-1) r)
    in (p,t+1,ins p l);;

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

Exemples (list3.ml)

```
(* suppression : temps lineaire en la position p *)
let delete_at_point (p,t,l) =
    let rec del p = function
      | [] -> assert false
      | h::r -> if p=0 then r else h::(del (p-1) r)
    in
    if p=t then failwith "fin de liste"
    else (p,t-1,del p l)
;;

y;;
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, où 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

- Essayons avec des champs mutables
- On définit un type d'enregistrements `'a cell` qui contient des champs pour :
  - le contenu d'une cellule
  - une "référence" (mutable) vers la cellule précédente
  - une "référence" (mutable) vers la cellule suivante.
- Les champs OCaml ne peuvent pas être NULL. On utilise donc des `('a cell option)` au lieu de `('a cell)` dans les champs.
- Note : on pouvait aussi utiliser des références OCaml `ref` plutôt que rendre les champs mutables.

Exemples (list4.ml)

```
(* careful, danger ahead, mutability ! *)
type 'a cell = { info:'a;
                 mutable next:'a cell option;
                 mutable prev:'a cell option
               };;

(* la bibliotheque standard definit :
type 'a option = None | Some of 'a
 *)
```

```
(* insertion, temps constant *)
let insert a = function
  | None -> Some {info=a; prev=None; next=None}
  | Some c as here ->
    let p = c.prev in
    let c' = Some {info=a; prev=p; next=here} in
    c.prev <- c';
    begin match p with
    | None -> ()
    | Some cp -> cp.next <- c'
    end;
    c'
;;
let l = insert 1 None ;;
let l'= insert 2 l ;; (* cycles ! *)
```

```
let rec of_list = function
  | [] -> None
  | a::r -> (* insert a (of_list r) *) (* ou bien: *)
     let o = of_list r in
     let c = Some {info=a; prev=None; next=o} in
     match o with
      | None -> c
      | Some c' -> c'.prev <- c; c ;;
let rec toutagauche = function
  | None -> None
  | Some {prev=None} as c -> c
  | Some {prev=c} -> toutagauche c ;;
let to_list o =
  let rec build acc = function
     | None -> List.rev acc
     | Some {info=a;next=c} -> build (a::acc) c
  in build [] (toutagauche o) ;;
```

Exemples (list7.ml)

```
(* deplacement, temps constant *)
let agauche = function
    | None -> failwith "Liste vide"
    | Some {prev=None} -> failwith "Deja a gauche"
    | Some {prev=c} -> c
;;
let adroite = function
    | None -> failwith "Liste vide"
    | Some {next=None} -> failwith "Deja a droite"
    | Some {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

- affichage problématique dans le toplevel (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

Ceci est un zipper de listes !

Exemples (ziplist1.ml)

```
type 'a pile = 'a list
type 'a listzipper = 'a pile * 'a list

(* navigation, 'a listzipper -> 'a listzipper *)
let agauche = function
  | ([],_)    -> failwith "Deja a gauche"
  | (a::p,l) -> (p, a::l);;

let adroite = function
  | (p,[])    -> failwith "Deja a droite"
  | (p,a::l) -> (a::p, l);;

(* chaque mouvement est en temps constant *)
```

Exemples (ziplist2.ml)

```
(* conversions *)
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
 (* ou List.rev_append c l *)
;;

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

Exemples (ziplist3.ml)

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

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

from_list ['a'; 'b'; 'd'; 'e']
  |> adroite |> adroite |> insert 'c' |> to_list ;;
```

Zipper des arbres binaires

- Prenons des 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 (stockant le côté où on n'est *pas* descendu)

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;;

(* NB pour les listzipper, un bloc = 'a *)
```

Exemples (zipbintree2.ml)

```
let bas_a_gauche : 'a arbrezipper -> 'a arbrezipper
= function
| pile,Feuille -> failwith "Feuille"
| pile,Noeud (x,g,d) -> (Gauche, x, d)::pile, g;;

let bas_a_droite : 'a arbrezipper -> 'a arbrezipper
= function
| pile,Feuille -> failwith "Feuille"
| pile,Noeud (x,g,d) -> (Droite, x, g)::pile, d;;

let en_haut : 'a arbrezipper -> 'a arbrezipper
= function
| (Gauche,x,t)::p, arbre -> p, Noeud (x,arbre,t)
| (Droite,x,t)::p, arbre -> p, Noeud (x,t,arbre)
| [], _ -> failwith "Racine";;
```

L'intuition derrière les zippers en général

- Une paire consistant en la sous-structure qu'on a selectionnée, et le *contexte* de la sous-structure.
- Le contexte est une liste de blocs, utilisée comme une pile : l'ordre à l'envers ("inside-out") est l'ordre naturel pour reconstruire la structure complète à partir de la sous-structure.
- Chaque bloc contient les éléments de la structure de données dans lesquels on n'est *pas* descendu, plus *un marqueur* indiquant vers où on est descendu.
- Dans le cas des listes on n'avait pas besoin du marqueur car dans une liste on ne peut avancer que dans une seule direction.

Zipper et arbres *n*-aires

- Maintenant chaque nœud a une *liste* de fils, et les données sont sur les feuilles :

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

- On veut pouvoir naviguer d'un nœud vers son père, ou vers son fils le plus à gauche, puis vers ses frères à gauche et à droite.
- Pour naviguer dans la liste des frères on utilise la structure des zippers de 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
| (a::lp,l)::p, arbre -> (lp,arbre::l)::p,a
| ([],_)::p, _ -> failwith "Deja a gauche"
| _ -> failwith "Racine";;

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

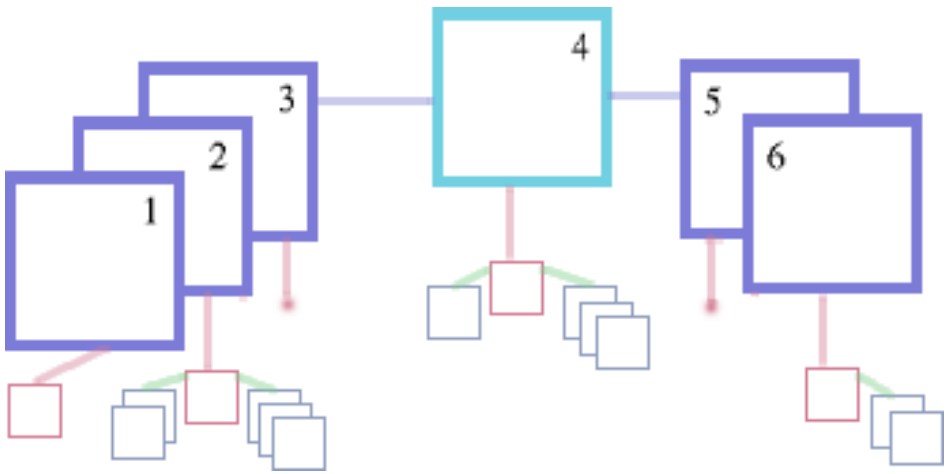
Exemples (ziptrees3.ml)

```
let en_bas : 'a narbrezipper -> 'a narbrezipper
= function
| pile, Noeud (filsg::freres) ->
    ([],freres)::pile, filsg
| _ -> failwith "Deja en bas";;

let en_haut : 'a narbrezipper -> 'a narbrezipper
= function
| (lp,l)::p, arbre ->
    p,Noeud(List.rev_append lp (arbre::l))
| [], _ -> failwith "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, où `x` est une variable prise dans un ensemble dénombrable

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

Une définition de type fonctionnels OCaml (ou Haskell ou ...) *sans types mutuels* 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 = Leaf | 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 = Leaf 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 :

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

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 $\mu x.1 + \text{'a} \times x$ et $F = 1 + \text{'a} \times x$
- on calcule la dérivée formelle $\partial_x F$ de F par rapport à x

  pour les listes, on a

  $$\partial_x(1+\text{'a}\times x) = \partial_x 1+\partial_x(\text{'a}\times x) = 0+\text{'a}\times\partial_x x = \text{'a}\times 1 = \text{'a}$$
- 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 !

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 $\texttt{'a tree} + \texttt{'a tree}$)

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.