Plan du cours : Zippers

- Le problème : naviguer efficacement une structure de données
- Ce qui ne marche pas
- Ce qui marche : les Zippers de Huet
- Exemples
- Comment dériver des Zippers pour tout type de données
- Pour en savoir plus

Description des objectifs

Naviguer dans une structure de données (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

```
(* 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);;
val agauche : int * 'a * 'b -> int * 'a * 'b = <fun>

let adroite = function
        (p,t,l) when p=t -> failwith "fin de liste"
    |   (p,t,l) -> (p+1,t,l)
;;
val adroite : int * int * 'a -> int * int * 'a = <fun>
```


Des fonctions sur les listes avec position

```
(* 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-1,l));;
val insert_at_point : 'a -> int * int * 'a list -> int
*
*   <fun>
```

Des fonctions sur les listes avec position

```
(* 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 (p,t-1,del(p-1,l))
;;
val delete_at_point : int * int * 'a list -> int * int * 'a
|
```


Listes doublement chaînées

On peut essayer avec des références

```
type 'a dllist = Nil | Cell of 'a cell
and 'a cell = {info:'a;next:'a dllist ref ;prev:'a dllist ref};;

(* deplacement, temps constant *)
let agauche =
    function Nil -> failwith "Liste vide"
    | Cell {prev=c} when !c = Nil -> failwith "Deja a gauche"
    | Cell {prev=c} -> !c;;
val agauche : 'a dllist -> 'a dllist = <fun>

(* insertion, temps constant *)
let insert a =
    function Nil -> Cell {info=a;prev=ref Nil;next=ref Nil}
    | Cell c as dll ->
          let c' = Cell {info=a;prev=ref !(c.prev);next=ref dll} in
          c.prev:=c'; c'
;;
val insert : 'a -> 'a dllist -> 'a dllist = <fun>
```


Listes doublement chaînées

Ce n'est pas idéal

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

Zipper de listes

Gérard Huet 1997

```
(* 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;;

let a_gauche : 'a listzipper -> 'a listzipper = function
| ([],_)   -> raise (Zipper "Deja a gauche")
| (a::p,l) -> (p, a::l);;
val a_gauche : 'a listzipper -> 'a listzipper = <fun>

let a_droite : 'a listzipper -> 'a listzipper  = function
| (p,[])   -> raise (Zipper "Deja a droite")
| (p,a::l) -> (a::p, l);;
val a_droite : 'a listzipper -> 'a listzipper = <fun>
```

Zipper de listes

```
let insert v  : 'a listzipper -> 'a listzipper = function
   (pile,liste) -> (pile, v::liste);;
|| val insert : 'a -> 'a listzipper -> 'a listzipper = <fun>
```

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


L'intuition derrière les zippers

Considérons une visite récursive d'une liste

```
let rec visit = function
|[] -> ()
| a::r -> visit r
;;
|| val visit : 'a list -> unit = <fun>
```

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.

Zipper des arbres binaires

```
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;;
```

Zipper des arbres binaires

```
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;;
val bas_a_gauche : 'a arbrezipper -> 'a arbrezipper = <fun>
```

```
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;;
val bas_a_droite : 'a arbrezipper -> 'a arbrezipper = <fun>
```

```
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");;
val en_haut : 'a arbrezipper -> 'a arbrezipper = <fun>
```


Zipper des arbres n-aires

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

type 'a block = 'a narbre listzipper

type 'a pile = 'a block list

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


Zipper des arbres n-aires

```
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";;
val a_gauche : 'a narbrezipper -> 'a narbrezipper = <fun>
```

```
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");;
val a_droite : 'a narbrezipper -> 'a narbrezipper = <fun>
```

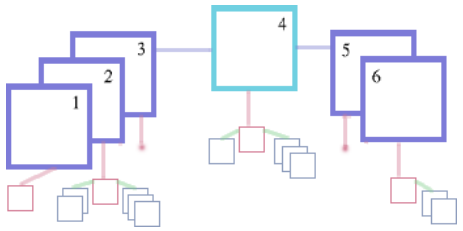

Zipper des arbres n-aires

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

```
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");;
val en_haut : 'a narbrezipper -> 'a narbrezipper = <fun>
```

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

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

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

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

```
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

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

c'est à dire

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

Rappel sur les dérivées des fonctions

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

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

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` !

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

Le cas des arbres n-aires

Le cas des arbres n-aires est plus complexe : nous devons traiter le type

$$\mu x.\texttt{'a} + x \ \textit{list} = \mu x.\texttt{'a} + (\mu y.1 + x \times y)$$

ce qui nous amène à calculer

$$\partial_x F = \partial_x(\texttt{'a} + (\mu y.1 + x \times y)) = 0 + \partial_x(\mu y.1 + x \times y)$$

Pour cela, McBride montre

$$\partial_x(\mu y.F) \quad = \quad \mu z.(\partial_x F)[y := \mu y.F] + (\partial_y F)[y := \mu y.F] \times z$$

ce qui permet d'obtenir, avec un peu d'effort, que le type du bloc de pile est

```
'a ntree list * 'a ntree list
```

mais on laissera cet aprofondissement aux plus aventureux.

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.