Le petit guide du bouturage
ou
comment écrire des structures mutables avec OCaml

Jean-Christophe Filliâtre
CNRS

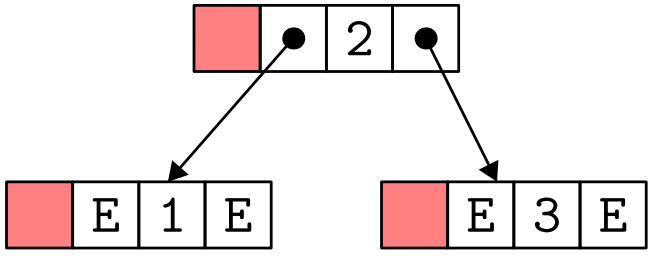
JFLA
8 janvier 2014

Caml, un bon langage pour l'algorithmique

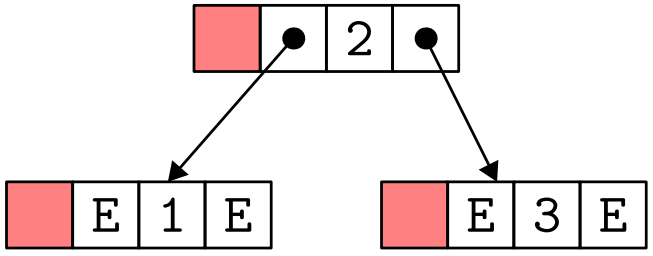
- types algébriques
- filtrage
- structures immuables (persistance facilitée)
- appel terminal bien compilé (récursivité facilitée)
- ordre supérieur
- polymorphisme, typage fort

- modèle d'exécution simple
- structures mutables

```
type tree = E | N of tree * int * tree
```



```
type tree = E | N of tree * int * tree
```




```
let rec min = function
  | E              -> assert false
  | N (E, x, _) -> x
  | N (l, _, _) -> min l
```

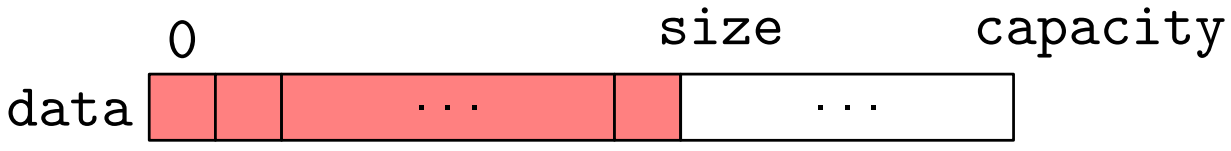
atout indéniable

"une fois le programme typé,
il ne reste que les erreurs algorithmiques"

une contrepartie néanmoins

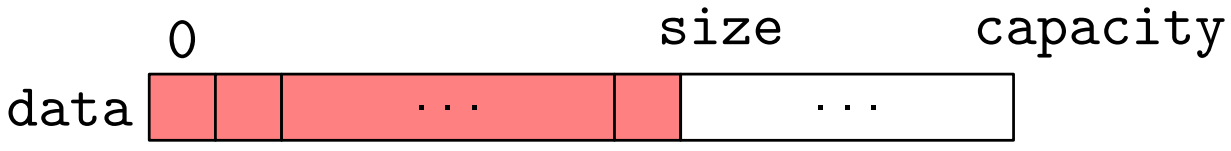
l'initialisation est obligatoire

```
type 'a vector = {
  mutable data: 'a array;
  mutable size: int;
}
```




```
let make capacity default = {
  data = Array.make capacity default;
  size = 0;
}
```

```
type 'a vector = {
  mutable data: 'a array;
  mutable size: int;
}
```




```
let make capacity default = {
  data = Array.make capacity default;
  size = 0;
}
```

une diminution de `size` ne doit pas empêcher le GC de réclamer les anciens éléments

on conserve la valeur `default`

```
type 'a vector = {
  mutable data: 'a array;
  mutable size: int;
        default: 'a;
}
```

on s'en sert pour effacer les anciennes valeurs

```
let resize v n =
  if n <= v.size then
    Array.fill v.data n (v.size - n) v.default
  else
    ...
```

(en Java, C, etc., on utiliserait `null`)

listes chaînées, arbres, etc.

on souhaite

- une solution naturelle, relativement efficace
- si possible conserver les bénéfices du filtrage

prenons l'exemple des arbres binaires

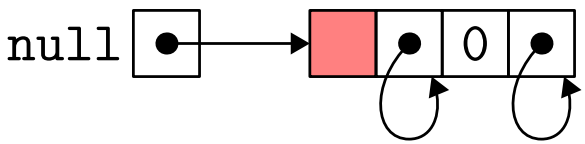
Algorithms, 4th edition
Sedgewick, Wayne.


```
static Tree add(Tree t, int x) {
  if (t == null) return new Tree(null, x, null);
  if      (x < t.elt) t.left  = add(t.left,  x);
  else if (x > t.elt) t.right = add(t.right, x);
  return t;
}
```

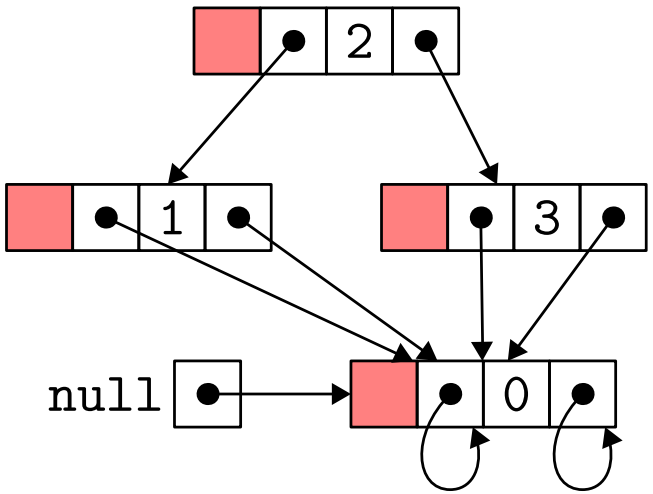
solution 1 : c'est vraiment `null`

```
type node = {
  v: int; mutable left: node; mutable right: node;
}
```

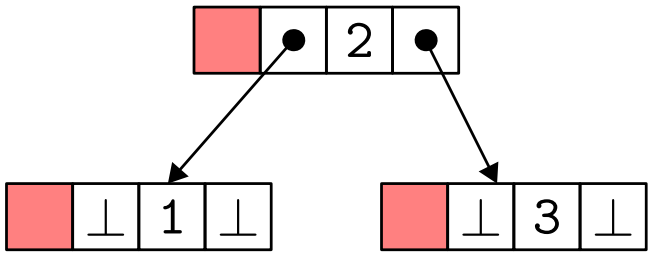
```
let rec null = { v = 0; left = null; right = null; }
```



```
let t1 = { left = null; v = 1; right = null; }
let t3 = { left = null; v = 3; right = null; }
let t2 = { left = t1;   v = 2; right = t3;   }
```



```
let t1 = { left = null; v = 1; right = null; }
let t3 = { left = null; v = 3; right = null; }
let t2 = { left = t1;   v = 2; right = t3;   }
```

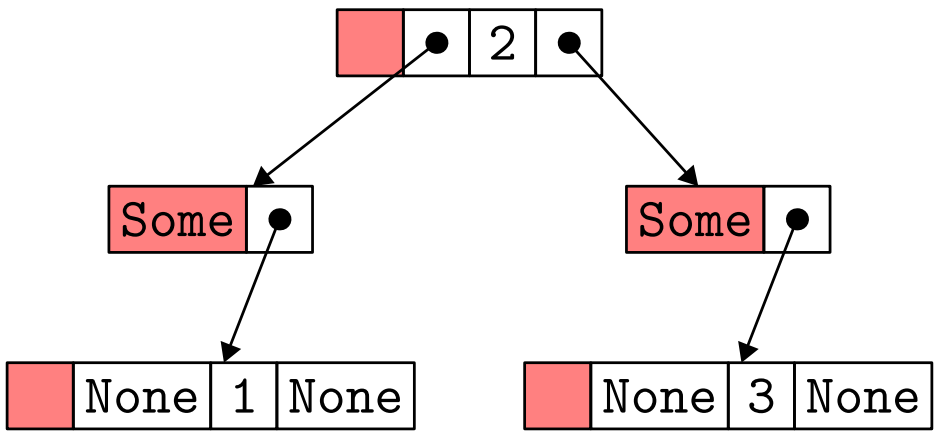


application (accidentelle) de = sur deux arbres

```
Out of memory during evaluation.
```

solution 2 : type option

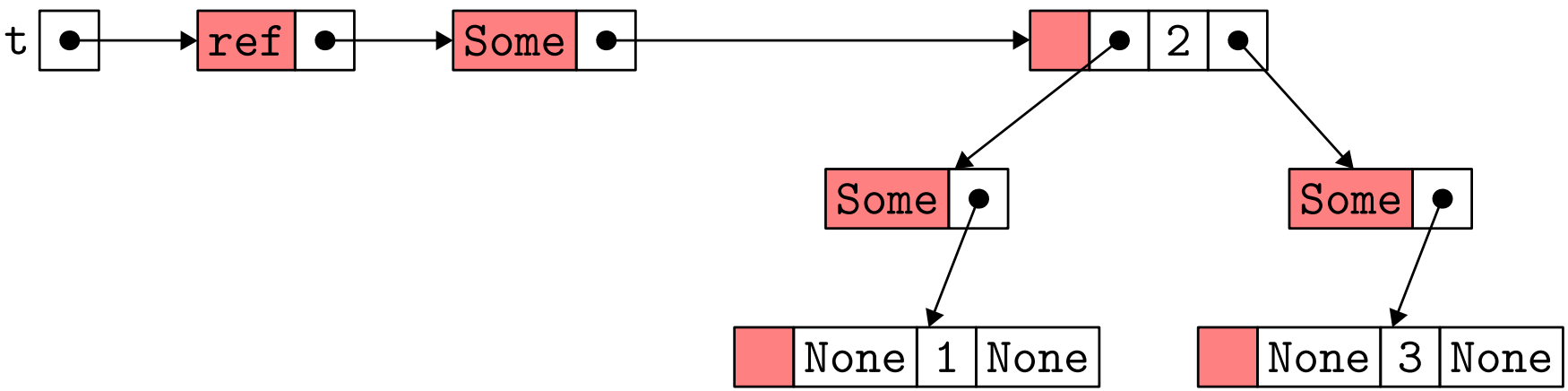
```
type node = {
                  v: int;
  mutable  left: node option;
  mutable right: node option;
}
```



```
let rec add x = function
  | None ->
      Some { v = x; left = None; right = None }
  | Some t as o ->
      if       x < t.v then t.left  <- add x t.left
      else if x > t.v then t.right <- add x t.right;
      o
```

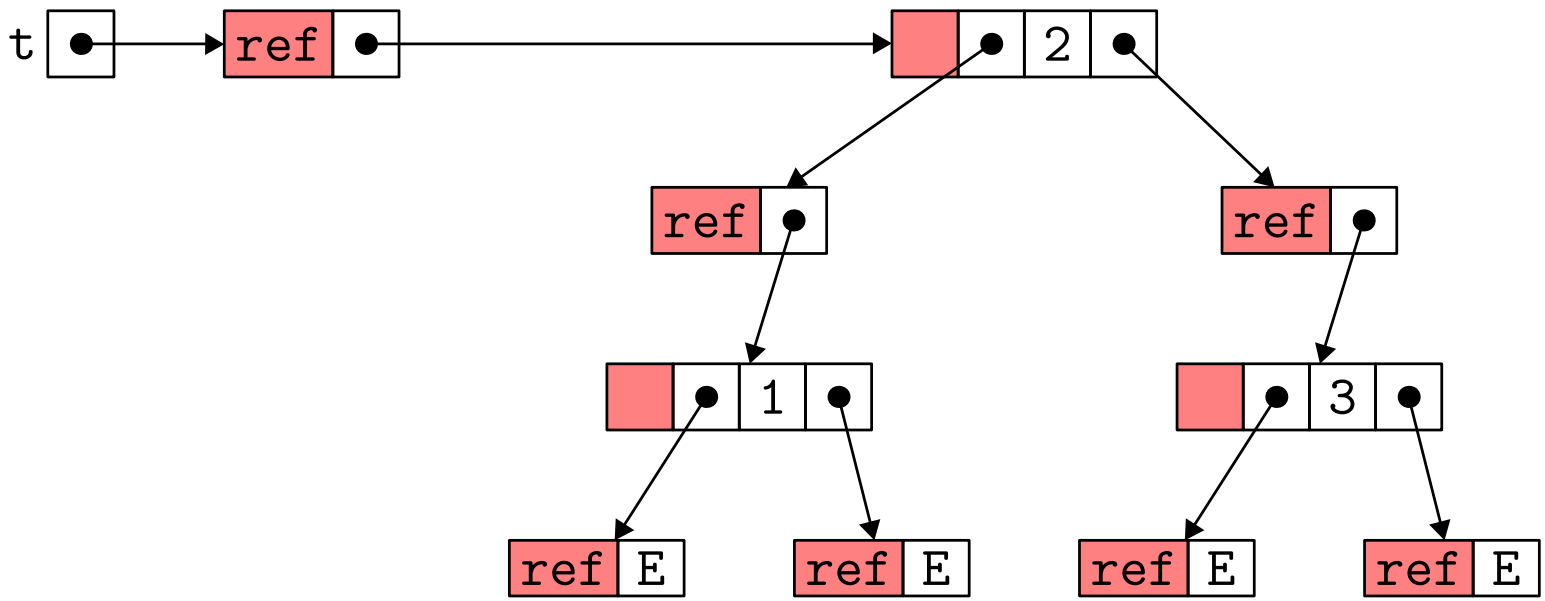
encapsulation à la racine

```
module S : SET = struct
  type t = node option ref
  let create () = ref None
  let add x s = s := add x !s
  ...
```



solution 3 : référence

```
type node = E | N of t * int * t
and  t    = node ref
```

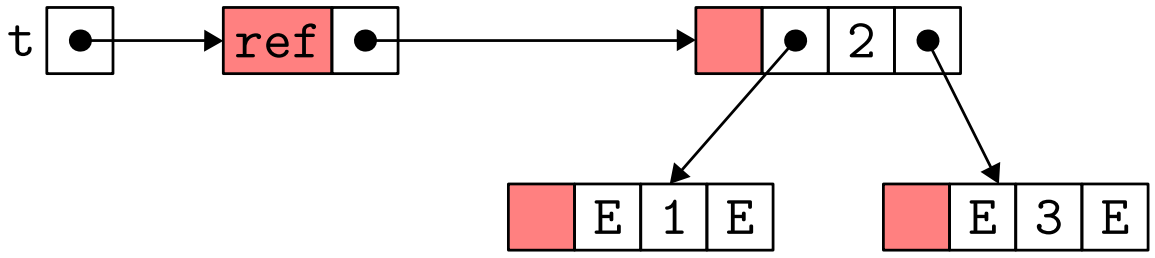


```
let rec add x t = match !t with
  | E ->
      t := N (ref E, x, ref E)
  | N (l, v, r) ->
      if x < v then add x l else if x > v then add x r
```

```
void add(Tree &t, int x) {
  if (t == NULL) t = new_tree(NULL, x, NULL);
  else if (x < t->elt) add(t->left, x);
  else if (x > t->elt) add(t->right, x);
}
```

solution 4 : le retour de Caml Light ?

```
type tree = E | N of mutable tree * int * mutable tree
```




```
let rec delete_min t = match t with
  | E              -> assert false
  | N (E, _, r) -> r
  | N (l, _, _) -> l <- delete_min l; t
```

avec OCaml, on doit tricher

```
type tree =
  | E
  | N of (*mutable*) tree * int * (*mutable*) tree
```

```
let set_left  (x: tree) (t: tree) =
  Obj.set_field (Obj.repr x) 0 (Obj.repr t)

let set_right (x: tree) (t: tree) =
  Obj.set_field (Obj.repr x) 2 (Obj.repr t)
```

comparaison des diverses solutions : espace

- occupation mémoire (en nombre de mots)
  pour un arbre binaire contenant N nœuds

- lignes de code pour un module fournissant
  `create`, `add`, `mem`, `remove`, `size`

	`null`	option	alg ref	alg mut	pure ref
occupation mémoire	$4N$	$6N$	$8N$	$4N$	$4N$
lignes de code	36	42	29	45	34

comparaison des diverses solutions : temps

	peigne 10k		aléatoire 1M parmi 10M			aléatoire 1M parmi 20k	
	add	mem	add	mem	rmv	add	rmv
`null`	1.320	0.596	**1.352**	0.503	**1.267**	0.427	0.023
option	2.168	0.695	4.339	0.523	2.579	1.067	0.028
alg ref	**0.611**	0.640	1.943	0.512	1.548	**0.324**	**0.012**
alg mut	1.340	0.576	1.436	0.499	1.347	0.451	0.020
pure ref	3.002	**0.568**	3.590	**0.497**	2.930	1.065	0.021

on peut écrire des structures mutables en OCaml

- en conservant les bénéfices des types algébriques (filtrage)
- avec l'élégance du passage par référence
- au prix d'une légère occupation mémoire supplémentaire

article complet dans mon bazar OCaml :

`https://www.lri.fr/~filliatr/software.fr.html`