# Gagner en passant à la corde

Jean-Christophe Filliâtre
CNRS

# Point de départ

dixième édition du concours de programmation de l'ICFP
manipulation de l'ADN d'un extra-terrestre (7,5 millions de bases)

on n'allait pas loin avec de simples chaînes de caractères
(quel que soit le langage)

il a fallu réinventer la roue

# La roue...

**Ropes : an Alternative to Strings**

Hans-J. Boehm, Russ Atkinson et Michael Plass

Software–Practice and Experience, vol. 25(12), 1315–1330 (December 1995)

(contexte : le développement du langage Cedar à Xerox PARC)

# Plan

1. structure de corde
2. plus loin avec les modules d'Ocaml

# Motivation

une bonne structure de chaînes de caractères :

- **immuable**
- **efficace**, notamment concaténation et extraction de sous-chaîne
- **sans limite de taille**, efficace sur les très longues chaînes

- un **fichier** doit pouvoir être considéré comme une chaîne de caractères

# Motivation

une bonne structure de chaînes de caractères :

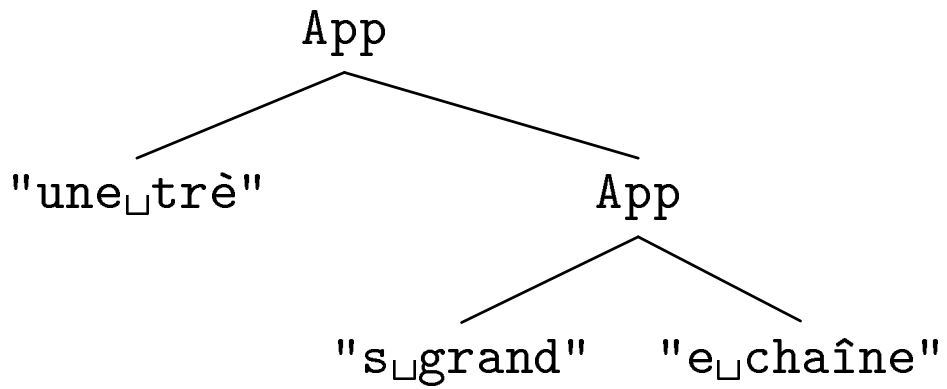
- **immuable**
- **efficace**, notamment concaténation et extraction de sous-chaîne
- **sans limite de taille**, efficace sur les très longues chaînes

- un **fichier** doit pouvoir être considéré comme une chaîne de caractères

# Idée de départ

une **corde** est un arbre binaire dont

- les nœuds représentent des concaténations
- les feuilles sont des chaînes usuelles



# Codage

```
type t =
  | Str of string
  | App of t × t
```

en pratique, on veut

- **partager des sous-chaînes** d'une même chaîne primitive
- obtenir la **longueur en temps constant**

```
type t =
  | Str of string × int × int (* offset, length *)
  | App of t × t × int        (* total length *)
```

# Codage

```
type t =
  | Str of string
  | App of t × t
```

en pratique, on veut

- **partager des sous-chaînes** d'une même chaîne primitive
- obtenir la **longueur en temps constant**

```
type t =
  | Str of string × int × int (* offset, length *)
  | App of t × t × int        (* total length *)
```

# Codage

```
type t =
  | Str of string
  | App of t × t
```

en pratique, on veut

- **partager des sous-chaînes** d'une même chaîne primitive
- obtenir la **longueur en temps constant**

```
type t =
  | Str of string × int × int (* offset, length *)
  | App of t × t × int        (* total length *)
```

# Opérations

**longueur**

```
val length : t → int (* O(1) *)
```

***i*-ième caractère**

```
val get : t → int → char
```

**sous-chaîne**

```
val sub : t → int → int → t
```

**concaténation**

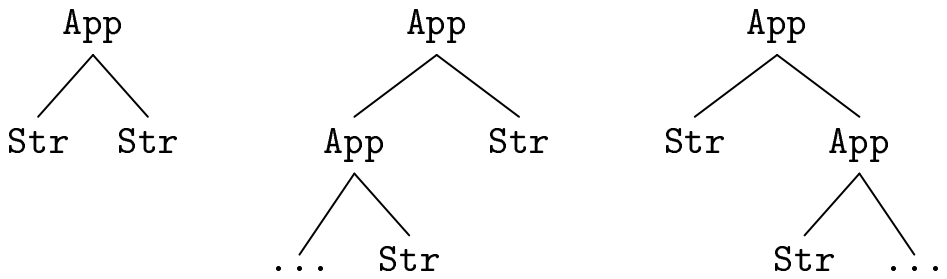
```
val append : t → t → t (* O(1) *)
let append t1 t2 = App (t1, t2, length t1 + length t2)
```

# Amélioration

en pratique

- on **concatène** effectivement les feuilles qui sont assez petites
- on cherche à **équilibrer** l'arbre

# Concaténation des petites feuilles




```
let append r1 r2 = match r1, r2 with
  | Str (s1, ofs1, len1), Str (s2, ofs2, len2)
      when len1 <= 256 && len2 <= 256 →
      Str (String.sub s1 ofs1 len1 ^ String.sub s2 ofs2 len2,
           0, len1 + len2)
  | ...
```

# Équilibrage

on peut équilibrer les cordes pendant la construction
(AVL, arbres rouges et noirs, etc.)

l'article original privilégie un rééquilibrage *a posteriori*

- quand la hauteur devient trop grande, ou
- quand l'utilisateur le demande

# Opérations de modification

**immédiates :**

```
let (++) = append

let delete t i = sub t 0 i ++ sub t (i+1) (length t-i-1)

let insert t i r = sub t 0 i ++ r ++ sub t i (length t - i)

let set t i c =
  sub t 0 i ++ singleton c ++ sub t (i+1) (length t-i-1)
```

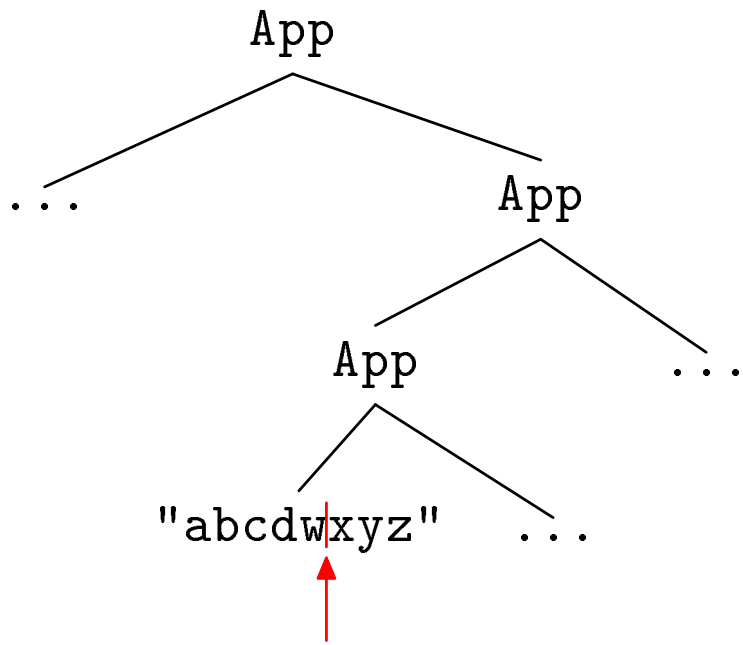
note : on peut optimiser avec une seule descente

# Curseurs

(mentionnés dans l'article de Boehm, Atkinson & Plass)

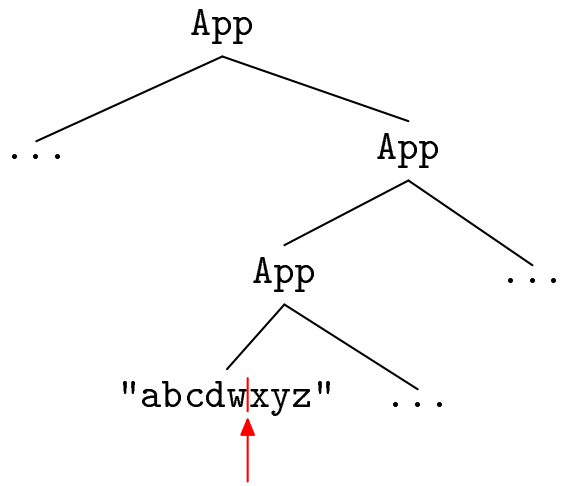
**idée :** se positionner en un point de la chaîne pour y effectuer des **modifications locales**, moins coûteuses que sur l'intégralité de la chaîne

c'est un **zipper** !



# Curseurs

```
type path =
  | Top
  | Left of path × t
  | Right of t × path
type cursor = {
  rpos : int;     (* relative position in the leaf *)
  lofs : int;     (* absolute position of the leaf *)
  leaf : t;       (* the leaf Str (str,ofs,len) *)
  path : path;    (* context = zipper *)
}
```




```
{ rpos = 4;
  lofs = ...;
  leaf = Str ("abcdwxyz", 0, 8);
  path = Left (Left (Right (..., Top),
                           ...),
                     ...)
}
```

# Pour finir...

l'article contient également

- des détails concernant la réalisation dans Cedar
- des tests de performance

# plus loin avec les modules d'Ocaml

# Foncteur naturel

```
module type STRING = sig
  type t
  type char
  val length : t → int
  val empty : t
  val singleton : char → t
  val get : t → int → char
  val append : t → t → t
  val sub : t → int → int → t
end

module Rope(S : STRING) : STRING with type char = S.char
```

peu importe le type `char`

# Une infinité de structures de données

```
module R1 = Rope(String)
```

$\Rightarrow$ des nœuds `Str` aux feuilles

```
module R2 = Rope(R1)
```

$\Rightarrow$ des nœuds `Str` à deux niveaux différents

```
module R3 = Rope(R2)
```

$\Rightarrow$ ...

# Tableaux redimensionnables

```
module Arr(T : sig type t end) = struct
  type char = T.t
  type t = char array
  let empty = [||]
  let singleton x = [|x|]
  include Array
end

module R = Rope(Arr(T))

type vector = R.t ref
```

# Autre application : fonctions vues comme des chaînes

dans l'article original, les feuilles peuvent aussi être des **fonctions**
(pour représenter la lecture dans un fichier, typiquement)

```
module SorF = struct
  type char = Char.t
  type t = S of string | F of int × (int → char)

  let length = function S s → String.length s | F (n,_) → n

  let get t i = match t with S s → s.[i] | F (_,f) → f i

  let sub t ofs len = match t with
    | S s → S (String.sub s ofs len)
    | F (n, f) → F (len, fun i → f (i - ofs))
  ...
end
module SF = Rope(SorF)
```

# Autre application : fonctions vues comme des chaînes

dans l'article original, les feuilles peuvent aussi être des **fonctions**
(pour représenter la lecture dans un fichier, typiquement)

```
module SorF = struct
  type char = Char.t
  type t = S of string | F of int × (int → char)

  let length = function S s → String.length s | F (n,_) → n

  let get t i = match t with S s → s.[i] | F (_,f) → f i

  let sub t ofs len = match t with
    | S s → S (String.sub s ofs len)
    | F (n, f) → F (len, fun i → f (i - ofs))
  ...
end
module SF = Rope(SorF)
```

# Dernière application : éditeur de texte

on veut écrire un éditeur de texte capable de supporter de **très gros** fichiers et/ou de **très grandes** lignes

idée : une corde de lignes pour le fichier, et une corde de caractères pour chaque ligne

bonus : la persistance de la structure de données facilite le retour en arrière (*undo*)

# Éditeur de texte

```
(* ligne = corde de caractères *)
module Line = Rope(String)

(* texte = corde de lignes *)
module Text = Rope(Arr(Line))

(* le texte en cours d'édition *)
let text = ref Text.empty
```

# Éditeur de texte

```
let insert_char l ofs c =
  let line = Text.get !text l in
  let line' = Line.insert line ofs c in
  text := Text.set !text l line'
```

```
let insert_newline l ofs =
  let line = Text.get !text l in
  let prefix = Line.sub line 0 ofs in
  let suffix = Line.sub line ofs (Line.length line - ofs) in
  let r = Text.insert !text l prefix in
  text := Text.set r (l + 1) suffix
```

# Éditeur de texte

```
let insert_char l ofs c =
  let line = Text.get !text l in
  let line' = Line.insert line ofs c in
  text := Text.set !text l line'
```

```
let insert_newline l ofs =
  let line = Text.get !text l in
  let prefix = Line.sub line 0 ofs in
  let suffix = Line.sub line ofs (Line.length line - ofs) in
  let r = Text.insert !text l prefix in
  text := Text.set r (l + 1) suffix
```

# Éditeur de texte

| | | |
|---|---|---|
| Graphisme | 68 | ldc |
| Modèle | 55 | ldc |
| Vue | 54 | ldc |
| Contrôleur | 86 | ldc |
| total | **263** | ldc |

# Servez-vous...

`http://www.lri.fr/~filliatr/software.en.html`