HAL
archives-ouvertes.fr

Gagner en passant à la corde

Jean-Christophe Filliâtre

▶ **To cite this version:**

Jean-Christophe Filliâtre. Gagner en passant à la corde. JFLA (Journées Francophones des Langages Applicatifs), Jan 2008, Etretat, France. pp.139-152, 2008. <inria-00202841>

HAL Id: inria-00202841
https://hal.inria.fr/inria-00202841

Submitted on 8 Jan 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Gagner en passant à la corde

Jean-Christophe Filliâtre

CNRS
LRI, Université Paris Sud, 91405 Orsay, France
INRIA Futurs (ProVal), 91893 Orsay, France
filliatr@lri.fr

Résumé

Cet article présente une réalisation en Ocaml de la structure de cordes introduite par Boehm, Atkinson et Plass. Nous montrons notamment comment cette structure de données s'écrit naturellement comme un foncteur, transformant une structure de séquence en une autre structure de même interface. Cette fonctorisation a de nombreuses applications au-delà de l'article original. Nous en donnons plusieurs, dont un éditeur de texte dont les performances sur de très gros fichiers sont bien meilleures que celles des éditeurs les plus populaires.

1. Introduction

La dixième édition du concours de programmation de l'ICFP [4] a été l'occasion de découvrir ou de redécouvrir la structure de *cordes* (en anglais *ropes*). Il s'agit d'une structure de données pour les chaînes de caractères introduite par Boehm, Atkinson et Plass [3] dans le cadre du développement du langage Cedar à Xerox PARC. Dans cet article, les auteurs motivent l'introduction d'une structure de données alternative pour les chaînes de caractères par les arguments suivants :

- les chaînes doivent être *immuables* — autrement dit, elles doivent être réalisées par un type de données *persistant* [6] ;
- les opérations usuelles telles que la concaténation ou l'extraction d'une sous-chaîne doivent être *efficaces*, notamment en espace ;
- les chaînes ne doivent pas être limitées en taille, et les opérations doivent rester efficaces sur les très longues chaînes ;
- enfin, il doit être possible de considérer d'autres types de séquences de caractères, tels que des fichiers par exemple, comme des chaînes de caractères.

Il est clair que les chaînes de caractères fournies en standard dans tous les langages de programmation ne remplissent pas ces critères. Bien au contraire, elles peuvent être (et sont donc) modifiées en place, sont inefficaces quant à la concaténation et à l'extraction d'une sous-chaîne (car impliquant des copies) et enfin sont parfois sévèrement limitées en taille. Quant à la dernière fonctionnalité, elle n'est tout simplement jamais proposée.

En réponse à ces déficiences des chaînes de caractères usuelles, Boehm, Atkinson et Plass proposent la structure de *cordes*, où les chaînes sont représentées par des arbres binaires dont les nœuds représentent des concaténations et où les feuilles sont des chaînes de caractères usuelles. L'article décrit quelques algorithmes sur les cordes, dont un algorithme de rééquilibrage *a posteriori*, discute quelques points de réalisation dans les langages C et Cedar, et présente des tests de performances.

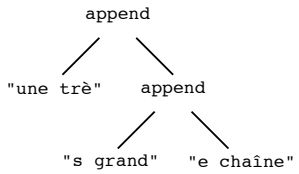
L'objectif du présent article est double. Tout d'abord, il s'agit de présenter une structure de données sans doute trop souvent négligée, ou simplement méconnue. En particulier, de nombreuses idées derrière la structure de cordes peuvent facilement être réutilisées dans d'autres contextes. Ensuite, nous allons au-delà de l'article original en réécrivant naturellement la structure de cordes comme un

foncteur. Celui-ci transforme une structure de séquence en une autre structure de même interface. Les applications de cette fonctorisation sont nombreuses et nous en décrivons plusieurs. En particulier, nous présentons la réalisation d'un mini éditeur de texte utilisant les cordes, dont les performances sont bien meilleures que celles de tous les éditeurs que nous avons testés sur des fichiers de très grande taille.

Cet article est organisé de la façon suivante. La section 2 décrit la structure de cordes et sa réalisation en Ocaml. La section 3 présente ensuite cette réalisation sous la forme d'un foncteur, dont les applications sont décrites dans la section 4. Enfin nous concluons avec des perspectives inspirées par cette réalisation des cordes. Cet article est illustré par du code écrit dans le langage Ocaml [1]. L'intégralité du code est disponible à l'adresse `http://www.lri.fr/~filliatr/software.fr.html`.

2. Réalisation des cordes en Ocaml

Cette section présente une réalisation des cordes en Ocaml. Sauf mention explicite du contraire, il s'agit d'idées présentes dans l'article original [3]. Comme nous l'avons expliqué dans l'introduction, une corde n'est rien d'autre qu'un arbre binaire dont les feuilles sont des chaînes (usuelles) de caractères, et dont les nœuds doivent être vus comme des concaténations. Ainsi, la corde



est l'une des multiples façons de représenter la chaîne `"une très grande chaîne"`. Une corde peut donc être directement codée par le type Ocaml suivant :

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

Cependant, deux considérations nous poussent à raffiner légèrement ce type. D'une part, de nombreux algorithmes ont besoin d'un accès efficace à la longueur d'une corde, notamment pour décider de descendre dans le sous-arbre gauche ou dans le sous-arbre droit d'un nœud `App`. Il est donc souhaitable d'ajouter la taille de la corde comme une décoration de chaque nœud interne. D'autre part, il est important de pouvoir partager des sous-chaînes entre les cordes elles-mêmes et avec les chaînes usuelles qui ont été utilisées pour les construire. Dès lors, plutôt que d'utiliser un nœud `Str` pointant sur une chaîne Ocaml complète, on va préférer un nœud désignant une sous-chaîne de cette chaîne Ocaml. On obtient donc le type suivant :

```
type rope =
  | Str of string × int × int
  | App of rope × rope × int
```

Un nœud `Str(`s`,`o`,`n`)` représente la sous-chaîne `s`$[o..o+n-1]$ c'est-à-dire la portion de la chaîne s de longueur n située au caractère o. Un nœud `App(`r_1`,`r_2`,`n`)` représente la concaténation des deux cordes r_1 et r_2, dont la longueur totale est n. (On aurait pu tout aussi bien stocker les tailles de r_1 et r_2 dans le nœud ; mais ne garder que la taille totale est plus économe en taille mémoire, sans perte d'efficacité en pratique.)

Dans un premier temps, nous présentons successivement des opérations d'accès, de construction et de modification sur les cordes. Puis nous montrons comment les cordes peuvent être équipées de curseurs afin d'améliorer l'efficacité de certaines opérations.

2.1. Opérations d'accès

L'accès à la longueur d'une corde est immédiat, par construction :

```
let length = function Str (_,_,n) | App (_,_,n) → n
```

L'accès à son i-ième caractère nécessite de descendre dans l'arbre jusqu'à la bonne feuille. Nous supposons ici que les caractères sont indexés à partir de 0, comme les chaînes usuelles d'Ocaml. Nous supposons d'autre part que la vérification de la validité de l'accès a été effectuée en amont. La partie récursive de l'accès s'écrit alors ainsi :

```
let rec get i = function
  | Str (s, ofs, _) →
      s.[ofs + i]
  | App (t1, t2, _) →
      let n1 = length t1 in if i < n1 then get i t1 else get (i - n1) t2
```

On voit ici l'intérêt d'obtenir la taille de `t1` en temps constant. La complexité de l'accès est donc bornée par la hauteur de l'arbre. Comme nous l'indiquerons dans la section suivante les cordes peuvent être équilibrées, afin de garantir un accès au pire logarithmique en fonction du nombre de nœuds.

2.2. Opérations de construction

La corde vide peut être représentée par une chaîne de longueur 0 :

```
let empty = Str ("", 0, 0)
```

La construction d'une corde à partir d'une chaîne Ocaml est immédiate :

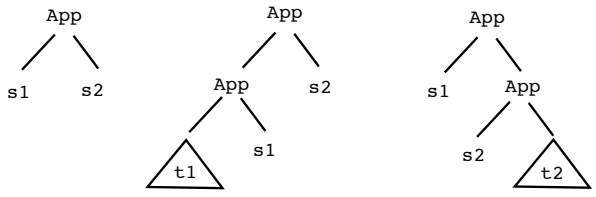
```
let of_string s = Str (s, 0, String.length s)
```

On notera cependant qu'afin de garantir le caractère immuable des cordes il faudrait en toute rigueur *copier* la chaîne `s` au lieu de simplement pointer vers celle-ci. Alors, et alors seulement, le caractère abstrait du type `rope` nous garantirait sa persistance.

Concaténation. La concaténation de deux cordes est *a priori* immédiate, le nœud `App` étant précisément là pour représenter une concaténation :

```
let mk_app t1 t2 = App (t1, t2, length t1 + length t2)
```

On note qu'il s'agit donc d'une opération en temps constant. Cependant, des concaténations massives vont amener le nombre de nœuds à croître rapidement, et donc la hauteur de l'arbre, au détriment des performances des autres opérations. Deux idées différentes permettent de maîtriser le nombre de nœuds et la hauteur de l'arbre. La première consiste à réaliser effectivement la concaténation des chaînes lorsque de petites feuilles se retrouvent côte à côte dans l'arbre. On peut choisir par exemple d'effectuer la concaténation des feuilles `s1` et `s2` dans les trois situations suivantes :



Ceci peut être effectué par l'opération de concaténation sur les cordes. Avec une longueur limite (arbitraire) de 256 caractères cela donne le code suivant :

```
let append t1 t2 = match t1, t2 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)
  | App (t1, Str (s1, ofs1, len1), _), Str (s2, ofs2, len2) when ... →
      App (t1, ...<idem>...)
  | Str (s1, ofs1, len1), App (Str (s2, ofs2, len2), t2, _) when ... →
      App (...<idem>..., t2)
  | _ →
      mk_app t1 t2
```

Comme on peut le constater, cette opération est particulièrement efficace lorsque l'on construit la corde par concaténations successives de petites chaînes à l'un de ses bouts, ce que l'on peut considérer comme une opération relativement courante.

La seconde amélioration à apporter à la construction des cordes pour en limiter la hauteur consiste naturellement à effectuer un *rééquilibrage* de l'arbre. Bien que l'article original évoque la possibilité d'un équilibrage incrémental, en utilisant des AVL par exemple, il privilégie un rééquilibrage *a posteriori*, effectué soit sélectivement lorsque la hauteur de l'arbre devient trop importante, soit explicitement à la demande de l'utilisateur. L'article décrit une méthode de rééquilibrage originale, basée sur la longueur des cordes plutôt que sur leur nombre de nœuds. Mais une méthode directe consistant à transformer l'arbre en un arbre binaire (presque) complet à partir de l'ensemble de ses feuilles est tout aussi efficace. Le code en est donné dans l'appendice A.

Extraction de sous-corde. L'extraction d'une sous-corde consiste à ne conserver que les parties de la corde impliquées dans la portion concernée, en modifiant éventuellement les nœuds `Str` se trouvant à cheval sur celle-ci. Si la sous-corde à extraire s'étend du caractère `start` (inclus) au caractère `stop` (exclu), et en supposant une fois encore les vérifications de validité déjà effectuées en amont, l'extraction peut s'écrire de la manière suivante :

```
let rec sub start stop = function
  | Str (s, ofs, _) →
      Str (s, ofs+start, stop-start)
  | App (t1, t2, _) →
      let n1 = length t1 in
      if stop <= n1 then sub start stop t1
      else if start >= n1 then sub (start-n1) (stop-n1) t2
      else mk_app (sub start n1 t1) (sub 0 (stop-n1) t2)
```

Dans le cas du nœud `App`, on distingue trois situations, suivant que la sous-corde se trouve dans `t1` (premier cas), dans `t2` (deuxième cas) ou bien à cheval sur `t1` et `t2` (troisième cas). Dans ce dernier cas, on réutilise la fonction `mk_app` ci-dessus qui construit le nœud `App` en calculant la longueur totale (*smart constructor*).

Ce code peut encore être légèrement amélioré pour partager les sous-arbres qui se trouvent être identiques entre la corde initiale et la sous-corde extraite. Pour cela, il suffit de retourner directement l'argument de `sub` dès lors que la portion à extraire constitue l'intégralité de la corde :

```
let rec sub start stop t =
  if start = 0 && stop = length t then t else ...<même code que ci-dessus>...
```

2.3. Opérations de modification

Les opérations de « modification » des cordes doivent respecter le caractère immuable de celles-ci, et donc renvoyer de nouvelles cordes. Pour insérer des caractères dans une corde, ou en supprimer, on peut donc simplement utiliser les opérations de concaténation et de sous-chaîne déjà écrites. Si on note la concaténation par l'opérateur infixe ++, alors on peut définir la suppression du i-ième caractère d'une corde ainsi :

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

De même, on peut définir l'insertion du caractère c juste après le i-ième caractère par

```
let insert t i c =
  sub t 0 i ++ of_string (String.make 1 c) ++ sub t i (length t - i)
```

on encore la modification du i-ième caractère en c par

```
let set t i c =
  sub t 0 i ++ of_string (String.make 1 c) ++ sub t (i + 1) (length t - i - 1)
```

Cependant, on note que ces fonctions descendent plusieurs fois dans l'arbre pour atteindre la même position. Il est donc préférable de les écrire de manière directe, avec un unique parcours de l'arbre. L'appendice B donne un tel code pour la fonction `delete`.

2.4. Curseurs

Il est fréquent de consulter ou de modifier une chaîne de manière répétée en un endroit précis, ou du moins sur un ensemble de positions proches les unes des autres. Dès lors, les descentes répétées vers le même endroit de la corde finissent par avoir un coût non négligeable par rapport à l'utilisation d'une chaîne de caractères usuelle. Pour y remédier, on peut ajouter aux cordes la notion de *curseur*, qui désigne une position dans la corde. Cette notion est présente dans l'article original et quelques détails de codage sont évoqués. Bien qu'il ne soit pas mentionné explicitement, on reconnaît là la description d'un *zipper* [5]. En effet, on peut facilement représenter un curseur par la donnée d'une feuille de la corde, d'un indice dans cette feuille et du contexte dans lequel apparaît cette feuille. Ce dernier élément peut être représenté simplement par le chemin menant de la feuille à la racine de la corde :

```
type path =
  | Top
  | Left  of path × rope
  | Right of rope × path
```

Le curseur est alors un enregistrement contenant l'ensemble des informations le définissant :

```
type cursor = {
  rpos: int;    (* position relative dans la feuille *)
  lofs: int;    (* position absolue de la feuille *)
  leaf: rope;   (* la feuille, de la forme Str (s,ofs,len) *)
  path: path;   (* le zipper *)
}
```

L'intérêt des curseurs est alors clair : des modifications locales au point désigné par le curseur peuvent maintenant être effectuées en temps constant, là où elles nécessitaient auparavant un temps proportionnel à la hauteur de la corde. Ainsi l'accès au caractère désigné par la curseur est immédiat :

```
let cursor_get c = let Str (s, ofs, len) = c.leaf in s.[ofs + c.rpos]
```

De même on peut écrire des fonctions d'insertion ou de suppression opérant sur un curseur en temps constant.

Passer de la corde au curseur et réciproquement se fait en utilisant les opérations usuelles d'ouverture et de fermeture du *zipper*. Ainsi la fonction `cursor_at` construit un curseur placé au caractère `i` de la corde `r`. Il s'agit de descendre jusqu'à la feuille concernée, tout en maintenant le contexte courant (argument `p` de `zip`) et la position courante dans la corde toute entière (argument `lofs`) :

```
let cursor_at r i =
  let rec zip lofs p = function
    | Str (_, _, len) as leaf  →
        { rpos = i - lofs; lofs = lofs; leaf = leaf; path = p }
    | App (t1, t2, _) →
        let n1 = length t1 in
        if i < lofs + n1 then
          zip lofs (Left (p, t2)) t1
        else
          zip (lofs + n1) (Right (t1, p)) t2
  in
  if i < 0 || i > length r then raise Out_of_bounds;
  zip 0 Top r
```

La fonction inverse reconstruit une corde à partir du contexte, en utilisant la fonction `mk_app` :

```
let rope_of_cursor c =
  let rec unzip t = function
    | Top → t
    | Left (p, tr) → unzip (mk_app t tr) p
    | Right (tl, p) → unzip (mk_app tl t) p
  in
  unzip c.leaf c.path
```

Le coût de ces deux opérations est proportionnel à la hauteur de l'arbre.

3. Fonctorisation

Jusqu'à présent, nous nous sommes contentés de coder en Ocaml les idées présentes dans l'article de Boehm, Atkinson et Plass [3]. Nous allons maintenant montrer comment la puissance du langage Ocaml peut être mise à profit pour aller plus loin. La première étape consiste à réécrire le code ci-dessus sous la forme d'un foncteur.

En effet, nous avons utilisé le type primitif de chaînes de caractères d'Ocaml pour les feuilles de nos cordes, mais cela n'avait rien d'une nécessité. Nous aurions pu tout aussi bien utiliser des tableaux ou des listes de caractères, et plus généralement encore des séquences de valeurs d'un autre type que celui des caractères. Plus généralement, il suffit de disposer d'une structure de données pour des *séquences* d'un certain *type de caractères*. On peut alors construire des cordes dont les feuilles seront réalisées par ces séquences-là. La structure de données obtenue a *la même signature* que celle de départ, à savoir celle de séquences pour le même type de caractères. Les cordes sont donc naturellement un foncteur transformant une structure de séquence en une autre structure de séquence de même interface.

Plus précisément, on peut se donner l'interface minimale suivante pour les structures de séquence :

```
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
```

où `t` est le type des séquences et `char` le type de ses caractères. Les cordes sont alors un foncteur ayant le profil suivant :

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

c'est-à-dire transformant un module `S` d'interface `STRING` en un module de même interface pour les mêmes caractères. Le codage de ce foncteur est immédiat, et en tout point semblable à ce que nous avons décrit dans la section précédente :

```
module Rope(S : STRING) = struct
  type t =
    | Str of S.t × int × int
    | App of t × t × int
  ...
end
```

En particulier, si on applique ce foncteur au module `String` de la bibliothèque standard d'Ocaml, on retrouve exactement le code décrit précédemment[1].

En pratique, le foncteur `Rope` fournit plus d'opérations en sortie qu'il n'en prend en entrée. On trouve en effet les opérations de modification dans le module résultat, alors qu'elles ne sont pas nécessaires dans le module d'entrée `S`. On note également que les deux opérations `append` et `sub` ne sont nécessaires dans `S` que si on souhaite effectuer l'optimisation décrite section 2.2 consistant à concaténer les petites feuilles.

4. Des avantages de la fonctorisation

Dans cette section nous présentons les avantages de la fonctorisation décrite dans la section précédente, au travers de plusieurs applications.

4.1. Itérations répétées du foncteur

Comme nous l'avons déjà dit, appliquer le foncteur `Rope` au module `String` nous redonne le codage des cordes présenté section 2, et le module obtenu a notamment une interface compatible avec celle de `String`. Dès lors, on peut appliquer de nouveau le foncteur `Rope`, pour obtenir un nouveau module de cordes. En répétant le processus, on obtient une infinité de structures de cordes :

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

[1] Le module `String` de la bibliothèque standard d'Ocaml ne définit pas de type `char` et il faut donc enrober `String` dans un module ajoutant `type char = Char.t`.

```
module R2 = Rope(R1)
module R3 = Rope(R2)
...
```

En quoi ces différentes structures de données diffèrent-elles ? Nous connaissons déjà `R1`. Les cordes de `R2` sont des arbres dont les feuilles sont des (sous-)cordes de `R1`, donc des arbres possédant un nœud « sous-corde » entre leur racine et leurs vraies feuilles, c'est-à-dire les chaînes Ocaml. De même, les cordes de `R3` sont des arbres dont les feuilles sont des cordes de `R2`, donc des arbres possédant deux nœuds « sous-corde » entre leur racine et leurs vraies feuilles, etc. Selon la stratégie adoptée concernant la concaténation des petites feuilles, on a donc des suspensions de calcul de sous-cordes au milieu des arbres, dont le nombre maximal est déterminé par le nombre d'applications successives du foncteur `Rope`. Le code disponible en ligne propose d'ailleurs un foncteur prenant un second argument pour contrôler les stratégies de concaténation des feuilles et de rééquilibrage.

Il est naturel de considérer le point-fixe obtenu en appliquant une infinité de fois le foncteur. Le type Ocaml résultant est le suivant :

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

Sans surprise, c'est le type d'un arbre où l'on peut suspendre aussi bien des opérations de concaténation que des calculs de sous-cordes. Tout dépend ensuite de la stratégie choisie pour effectuer tout de suite, ou bien au contraire suspendre, les opérations de concaténation et de sous-corde. (Nous n'avons pas mené d'expérience pour comparer de telles stratégies.)

4.2. Tableaux

Dans le foncteur `Rope` le type des caractères n'est pas fixé, et il n'y a donc pas lieu de se limiter au type primitif `char`. Les cordes constituent donc autant une structure de tableaux que de chaînes de caractères. Pour obtenir des tableaux dont les éléments sont de type `elt` il suffit d'instancier le foncteur `Rope` avec le paramètre suivant :

```
module S = struct
  type char = elt
  type t = elt array
  let length = Array.length
  let empty = [||]
  let singleton l = [|l|]
  let get = Array.unsafe_get
  let append = Array.append
  let sub = Array.sub
end
```

Bien entendu, les tableaux obtenus n'ont pas l'efficacité des tableaux primitifs en ce qui concerne l'accès et la mise à jour. En revanche, ils offrent tous les avantages des cordes : persistance, opérations `append` et `sub` efficaces, et virtuellement aucune limite de taille. En particulier, ces cordes-tableaux permettent aisément de réaliser des tableaux redimensionnables, un problème récurrent avec Ocaml, où la limite de taille des tableaux est particulièrement basse sur les machines 32 bits (à savoir 4194303). Nous utiliserons de tels tableaux dans la section 4.4.

4.3. Fonctions vues comme des chaînes

L'article original introduisant les cordes insiste sur la possibilité pour une corde d'avoir des feuilles réalisées non pas par des chaînes usuelles mais par des *fonctions*. L'idée est de pouvoir représenter par exemple le contenu d'un très gros fichier, sans avoir à le charger en mémoire. La structure de cordes fournie dans la bibliothèque STL [2] reprend cette fonctionnalité.

Si nous n'avons pas inclus cette possibilité dans notre codage initial, c'est parce qu'il est aisé de l'obtenir *a posteriori* grâce à notre foncteur. En effet, puisque nous pouvons fixer le type de feuilles, nous pouvons choisir un type offrant l'alternative entre chaînes de caractères et fonctions :

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

Une feuille de la forme `F(`n`,`f`)` représente une chaîne de longueur n dont le i-ième caractère est $(f\ i)$. Il est alors facile de réaliser les différentes opérations exigées sur ce type somme :

```
  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
```

4.4. Un éditeur de texte

Pour terminer, nous présentons une application réaliste de la structure de cordes, à savoir un éditeur de texte. Si un éditeur de texte jouet peut être réalisé directement en utilisant les chaînes de caractères fournies par le langage, un éditeur réaliste se doit d'utiliser une structure adaptée aux fichiers de grande taille. Il est d'ailleurs frappant de constater que, malgré leurs efforts dans ce sens, les éditeurs populaires tels que Emacs ou vi montrent rapidement leurs limites sur un fichier contenant une unique ligne de plus de 7 millions de caractères (tel que le code ADN constituant le point de départ du dernier concours de programmation de l'ICFP [4]), que ce soit pour l'insertion, la suppression ou la recherche.

L'application de la structure de cordes aux éditeurs de texte est déjà mentionnée dans l'article de Boehm, Atkinson et Plass. Mais là où les auteurs suggèrent d'utiliser une corde pour représenter l'intégralité du texte en cours d'édition, nous nous proposons d'utiliser notre foncteur pour encore plus de flexibilité. En effet, une unique corde rend fastidieuse la gestion des différentes lignes du texte (il faut rechercher les sauts de ligne, ou maintenir leur position dans une table correctement synchronisée). Au lieu de cela, nous pouvons utiliser une corde pour représenter l'ensemble des lignes du texte et représenter chaque ligne, c'est-à-dire chaque élément de cette corde, par une corde dont les éléments sont des caractères. Nous commençons donc par construire une structure de corde pour les lignes :

```
module Line = Rope(String)
```

puis une autre structure de corde pour le texte, comme un tableau de lignes à la manière de la section 4.2 :

```
module Text = Rope(struct
  type char = Line.t
  type t = Line.t array
  let empty = [||]
  ...
end)
```

Le texte manipulé par l'éditeur peut alors être stocké dans une unique référence contenant une corde de type `Text.t` :

```
let text = ref Text.empty
```

Pour insérer le caractère `c` à la position `ofs` dans la ligne `l` du texte, il suffit de récupérer cette ligne avec `Text.get`, de la modifier avec `Line.insert` et enfin de mettre à jour la ligne avec `Text.set` :

```
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'
```

De même pour supprimer un caractère :

```
let delete_char l ofs =
  let line = Text.get !text l ln in
  let line' = Line.delete line ofs in
  text := Text.set !text l line'
```

Enfin, pour insérer un retour-chariot à la position `ofs` dans la ligne `l` du texte, il suffit de couper la ligne `l` à la position `ofs` avec deux appels à `Line.sub`, puis d'insérer le préfixe à la ligne `l` avec `Text.insert` et de positionner le suffixe en ligne `l+1` avec `Text.set` :

```
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
```

On note qu'il n'est pas nécessaire ici de « décaler » toutes les lignes : on utilise l'insertion dans la corde des lignes, tout comme on a utilisé l'insertion dans les cordes de caractères pour `insert_char`.

Sur la base de ce code, nous avons codé un mini éditeur de texte, supportant les fonctionnalités suivantes : insertion de texte, suppression, déplacements élémentaires (flèches, début et fin de ligne, page suivante, page précédente) et recherche. La partie graphique utilise la bibliothèque SDL. Le reste du code suit le patron du *modèle-vue-contrôleur* [7] : le modèle est le contenu du fichier, dont la gestion est présentée ci-dessus ; la vue assure la correspondance entre le modèle et la portion qui en est affichée, ainsi que la position du curseur graphique ; enfin le contrôleur est la boucle d'interaction gérant les événements clavier. Les lignes de code sont ainsi partagées :

module	**lignes**
graphisme	68
modèle	55
vue	54
contrôleur	86
total	**263**

Les résultats sont conformes aux attentes les plus optimistes : l'édition est absolument fluide quelle que soit la taille du fichier et quelle que soit la taille des lignes (nous avons fait des tests sur des fichiers de plus de 200 Mo et sur des lignes de plus de 8 Mo). Le code de ce mini-éditeur n'utilise même pas les curseurs présentés section 2.4, pour plus de simplicité. On peut donc espérer d'encore meilleurs résultats pour un éditeur utilisant un curseur (au sens des cordes) pour représenter la position actuelle du curseur graphique, et donc une insertion plus efficace. Enfin, il est important de noter que la persistance de la structure de corde est un avantage supplémentaire dans un logiciel tel qu'un éditeur de texte, car permettant une réalisation quasi-triviale du retour en arrière (*undo*).

5. Conclusion

Dans cet article, nous avons cherché à populariser une structure de données injustement méconnue, à savoir les *cordes*. Suivant le proverbe qui dit qu'il faut avoir plusieurs cordes à son arc, nous avons écrit cette structure de données comme un foncteur paramétré par la structure de séquences sous-jacente. Cette fonctorisation ouvre de multiples perspectives à l'utilisation des cordes, bien au-delà de ce qui est suggéré dans l'article original les introduisant. Nous avons notamment montré comment cette généricité permet de jeter facilement les bases d'un éditeur de texte à même d'appréhender des fichiers de très grande taille.

La réalisation de cette structure de cordes a permis de soulever au moins deux questions méritant plus ample investigation. D'une part, le type obtenu par point fixe section 4.1 constitue une variante des cordes où l'opération de sous-corde peut être suspendue au même titre que l'est celle de concaténation. Pour en tirer tous les avantages, il faudrait étudier les conditions sous lesquelles il est intéressant d'effectuer cette suspension, ou au contraire de ne pas l'effectuer. Il est clair par exemple que lors de plusieurs extractions successives de sous-cordes, les premières ont intérêt à être suspendues.

La seconde interrogation qui vient naturellement à l'esprit consiste à se demander si l'idée de suspendre des opérations en les représentant sous forme d'un arbre peut être généralisée avec profit à d'autres structures de données. On peut ainsi imaginer une structure d'ensembles où une opération telle que l'union, par exemple, n'est pas systématiquement effectuée. Restent à déterminer les conditions sous lesquelles une telle suspension peut être asymptotiquement avantageuse.

Références

[1] Le langage Objective Caml. `http://caml.inria.fr/`.

[2] SGI's extension to the C++ Standard Template Library : Ropes. `http://www.sgi.com/tech/stl/Rope.html`.

[3] Hans-Juergen Boehm, Russell R. Atkinson, and Michael F. Plass. Ropes : An alternative to strings. *Software - Practice and Experience*, 25(12) :1315–1330, 1995.

[4] Eelco Dolstra, Jur Hage, Bastiaan Heeren, Stefan Holdermans, Johan Jeuring, Andres Löh, Arie Middelkoop, Alexey Rodriguez, John van Schie, and Clara Löh. Morph Endo ! Report on the Tenth Interstellar Contest on Fuun Programming. Technical Report UU-CS-2007-029, Institute of Information and Computing Sciences, Utrecht University, 2007.

[5] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5) :549–554, Septembre 1997.

[6] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[7] Julien Signoles. Une approche fonctionnelle du modèle vue-contrôleur. In *Journées Francophones des Langages Applicatifs*, March 2005.

A. Rééquilibrage des cordes

Nous donnons ici un code simple pour rééquilibrer une corde *a posteriori*. Le code commence par construire la liste des feuilles (fonction `to_list`), tout en calculant le nombre de feuilles. Ensuite la fonction `build` reconstruit un arbre binaire équilibré, en coupant la liste en deux, récursivement.

```
let balance t =
  let rec to_list ((n, l) as acc) = function
    | Str _ as x → n + 1, x :: l
    | App (t1, t2, _) → to_list (to_list acc t2) t1
  in
  let rec build n l =
    assert (n >= 1);
    if n = 1 then begin
      match l with
        | [] → assert false
        | x :: r → x, r
    end else
      let n' = n / 2 in
      let t1, l = build n' l in
      let t2, l = build (n - n') l in
      mk_app t1 t2, l
  in
  let n, l = to_list (0, []) t in
  let t, l = build n l in
  assert (l = []);
  t
```

B. Suppression du i-ième caractère

La suppression d'un caractère dans une corde peut être optimisée en tenant compte des cas particuliers où ce caractère se trouve être au bord d'une feuille (à gauche ou à droite). La fonction `delete_rec` ci-dessous traite ces cas particuliers (trois premiers motifs), puis le cas général d'une feuille qu'il faut décomposer en deux nouvelles feuilles (où on notera le partage de la chaîne `s`) et enfin le cas d'un nœud de concaténation. Cette fonction suppose que la validité de l'indice `i` a été vérifiée en amont.

```
let rec delete_rec i = function
  | Str (_, _, 1) →
      assert (i = 0); empty
  | Str (s, ofs, len) when i = 0 →
      Str (s, ofs+1, len-1)
  | Str (s, ofs, len) when i = len-1 →
      Str (s, ofs, len-1)
  | Str (s, ofs, len) →
      mk_app (Str (s, ofs, i)) (Str (s, ofs+i+1, len-i-1))
  | App (t1, t2, _) →
      let n1 = length t1 in
      if i < n1 then
        mk_app (delete_rec i t1) t2
      else
```

```
      mk_app t1 (delete_rec (i-n1) t2)
```

On peut alors écrire la fonction `delete` qui effectue la vérification de validité avant d'appeler `delete_rec` :

```
let delete t i =
  if i < 0 || i >= length t then raise Out_of_bounds;
  delete_rec i t
```