Programmation fonctionnelle avancée

Notes de cours

Cours 4

30 septembre 2015

Sylvain Conchon

sylvain.conchon@lri.fr

Les cordes

Les cordes

On souhaite remédier aux problèmes suivants sur les chaînes de caractères :

- elles sont limitées en taille
- coût important de la concaténation et de l'extraction de sous-chaînes (notamment en espace)

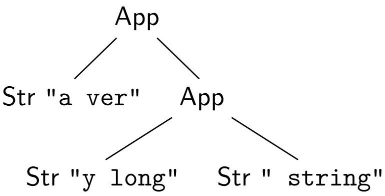
La structure de corde

Elle s'appuie sur la structure d'arbre binaire :

- les feuilles sont des chaînes de caractères
- les nœuds sont des concaténations

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

Exemple de corde pour représenter la chaîne
`"a very long string"`



Amélioration de la structure

- On stocke la taille dans les nœuds `App`
- On améliore le partage des sous-chaînes en précisant la partie partagée

```
type t =
  | Str of string * int * int
  | App of t * t * int
```

`Str(s, i, n)` correspond à la sous-chaîne `s[i..i+n-1]`

`App(r1, r2, n)` correspond à la concaténation de deux cordes `r1` et `r2` dont la longueur totale est `n`

Opérations élémentaires

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

let length = function
| Str (_, _, n)
| App (_, _, n) -> n

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

let make n c = of_string (String.make n c)
```

Accès au ième caractère

```
let rec unsafe_get t i = match t with
```

Accès au ième caractère

```
let rec unsafe_get t i = match t with
  | Str (s, ofs, _) ->
     s.[ofs + i]
```

Accès au ième caractère

```
let rec unsafe_get t i = match t with
  | Str (s, ofs, _) ->
    s.[ofs + i]
  | App (t1, t2, _) ->
    let n1 = length t1 in
    if i < n1 then
        unsafe_get t1 i
    else
        unsafe_get t2 (i - n1)
```

Accès au ième caractère

```
let rec unsafe_get t i = match t with
  | Str (s, ofs, _) ->
     s.[ofs + i]
  | App (t1, t2, _) ->
     let n1 = length t1 in
     if i < n1 then
         unsafe_get t1 i
     else
         unsafe_get t2 (i - n1)

let get t i =
  if i < 0 || i >= length t then invalid_arg "get";
  unsafe_get t i
```

Concaténation des cordes (1)

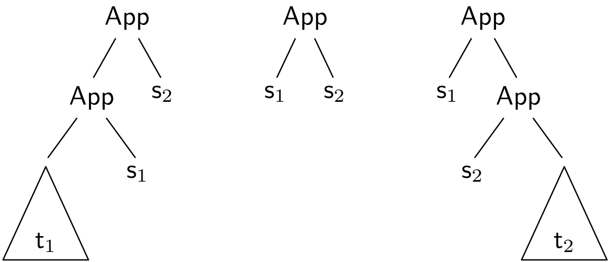
Elle pourrait être aussi simple que le code ci-dessous, mais cela ferait croître le nombre de nœuds trop rapidement

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

Concaténation des cordes (2)

Pour éviter de créer trop de nœuds, il faut concaténer les petites feuilles

Trois cas peuvent se présenter :



Concaténation des cordes (3)

On fixe la taille d'une petite feuille :

```
let small = 256
```

On se donne la fonction suivante pour concaténer deux fragments des chaînes s1 et s2

```
let append_string s1 ofs1 len1 s2 ofs2 len2 =
  let ss1 = String.sub s1 ofs1 len1 in
  let ss2 = String.sub s2 ofs2 len2 in
  Str (ss1 ^ ss2, 0, len1 + len2)
```

Concaténation des cordes (4)

```
let append t1 t2 = match t1, t2 with
```

Concaténation des cordes (4)

```
let append t1 t2 = match t1, t2 with
| Str (_,_,0), t | t, Str (_,_,0) ->
      t
```

Concaténation des cordes (4)

```
let append t1 t2 = match t1, t2 with
| Str (_,_,0), t | t, Str (_,_,0) ->
      t
| Str (s1, ofs1, len1), Str (s2, ofs2, len2)
  when len1 <= small && len2 <= small ->
    append_string s1 ofs1 len1 s2 ofs2 len2
```

Concaténation des cordes (4)

```
let append t1 t2 = match t1, t2 with
| Str (_,_,0), t | t, Str (_,_,0) ->
      t
| Str (s1, ofs1, len1), Str (s2, ofs2, len2)
  when len1 <= small && len2 <= small ->
    append_string s1 ofs1 len1 s2 ofs2 len2
| App (t1, Str (s1, ofs1, len1), _), Str (s2, ofs2, len2)
  when len1 <= small && len2 <= small ->
    App (t1, append_string s1 ofs1 len1 s2 ofs2 len2,
         length t1 + len1 + len2)
```

Concaténation des cordes (4)

```
let append t1 t2 = match t1, t2 with
| Str (_,_,0), t | t, Str (_,_,0) ->
       t
| Str (s1, ofs1, len1), Str (s2, ofs2, len2)
  when len1 <= small && len2 <= small ->
    append_string s1 ofs1 len1 s2 ofs2 len2
| App (t1, Str (s1, ofs1, len1), _), Str (s2, ofs2, len2)
  when len1 <= small && len2 <= small ->
    App (t1, append_string s1 ofs1 len1 s2 ofs2 len2,
         length t1 + len1 + len2)
| Str (s1, ofs1, len1), App (Str (s2, ofs2, len2), t2, _)
  when len1 <= small && len2 <= small ->
    App (append_string s1 ofs1 len1 s2 ofs2 len2, t2,
         len1 + len2 + length t2)
```

Concaténation des cordes (4)

```
let append t1 t2 = match t1, t2 with
| Str (_,_,0), t | t, Str (_,_,0) ->
       t
| Str (s1, ofs1, len1), Str (s2, ofs2, len2)
  when len1 <= small && len2 <= small ->
    append_string s1 ofs1 len1 s2 ofs2 len2
| App (t1, Str (s1, ofs1, len1), _), Str (s2, ofs2, len2)
  when len1 <= small && len2 <= small ->
    App (t1, append_string s1 ofs1 len1 s2 ofs2 len2,
         length t1 + len1 + len2)
| Str (s1, ofs1, len1), App (Str (s2, ofs2, len2), t2, _)
  when len1 <= small && len2 <= small ->
    App (append_string s1 ofs1 len1 s2 ofs2 len2, t2,
         len1 + len2 + length t2)
| t1, t2 ->
    App (t1, t2, length t1 + length t2)
```

Extraction de sous-corde (1)

```
let (++) = append
```

Extraction de sous-corde (1)

```
let (++) = append

let rec mksub start stop t =
```

Extraction de sous-corde (1)

```
let (++) = append

let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
```

Extraction de sous-corde (1)

```
let (++) = append

let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
  else match t with
```

Extraction de sous-corde (1)

```
let (++) = append

let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
  else match t with
    | Str (s, ofs, _) ->
      Str (s, ofs+start, stop-start)
```

Extraction de sous-corde (1)

```
let (++) = append

let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
  else match t with
    | Str (s, ofs, _) ->
      Str (s, ofs+start, stop-start)
    | App (t1, t2, _) ->
      let n1 = length t1 in
```

Extraction de sous-corde (1)

```
let (++) = append

let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
  else match t with
    | Str (s, ofs, _) ->
      Str (s, ofs+start, stop-start)
    | App (t1, t2, _) ->
      let n1 = length t1 in
      if stop <= n1 then
        mksub start stop t1
```

Extraction de sous-corde (1)

```
let (++) = append

let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
  else match t with
    | Str (s, ofs, _) ->
      Str (s, ofs+start, stop-start)
    | App (t1, t2, _) ->
      let n1 = length t1 in
      if stop <= n1 then
        mksub start stop t1
      else if start >= n1 then
        mksub (start-n1) (stop-n1) t2
```

Extraction de sous-corde (1)

```
let (++) = append

let rec mksub start stop t =
  if start = 0 && stop = length t then
    t
  else match t with
    | Str (s, ofs, _) ->
      Str (s, ofs+start, stop-start)
    | App (t1, t2, _) ->
      let n1 = length t1 in
      if stop <= n1 then
        mksub start stop t1
      else if start >= n1 then
        mksub (start-n1) (stop-n1) t2
      else mksub start n1 t1 ++ mksub 0 (stop-n1) t2
```

Extraction de sous-corde (2)

```
let sub t ofs len =
  let stop = ofs + len in
```

Extraction de sous-corde (2)

```
let sub t ofs len =
  let stop = ofs + len in
  if ofs < 0 || len < 0 || stop > length t then
      invalid_arg "sub";
```

Extraction de sous-corde (2)

```
let sub t ofs len =
  let stop = ofs + len in
  if ofs < 0 || len < 0 || stop > length t then
      invalid_arg "sub";
  if len = 0 then
    empty
```

Extraction de sous-corde (2)

```
let sub t ofs len =
  let stop = ofs + len in
  if ofs < 0 || len < 0 || stop > length t then
      invalid_arg "sub";
  if len = 0 then
    empty
  else mksub ofs stop t
```

Opérations de modification sur les cordes

Très facile à écrire en utilisant sub

```
let set t i c =   let n = length t in
  if i < 0 || i >= n then invalid_arg "set";
  sub t 0 i ++ make 1 c ++ sub t (i + 1) (n - i - 1)
```

Opérations de modification sur les cordes

Très facile à écrire en utilisant sub

```
let set t i c =   let n = length t in
  if i < 0 || i >= n then invalid_arg "set";
  sub t 0 i ++ make 1 c ++ sub t (i + 1) (n - i - 1)

let insert t i r =
  let n = length t in
  if i < 0 || i > n then invalid_arg "insert";
  sub t 0 i ++ r ++ sub t i (n - i)

let delete_char t i =
  let n = length t in
  if i < 0 || i >= n then invalid_arg "delete_char";
  sub t 0 i ++ sub§ t (i + 1) (n - i - 1)
```

Les arbres équilibrés

Recherche de l'équilibre

- La recherche dans un arbre ordonné est proportionnelle à la longueur de la plus grande branche de l'arbre
- Cette recherche est optimale pour des arbres de recherche équilibrés, c'est-à-dire les arbres de taille n et de hauteur $\log(\mathrm{n})$

Rééquilibrage des arbres de recherche

- L'efficacité de la recherche dans un arbre binaire ordonné dépend fortement de la forme de l'arbre
- La forme la pire est celle du peigne : un arbre où chaque nœud n'a qu'un seul successeur gauche ou droit (l'arbre n'est alors ni plus ni moins qu'une liste) ; l'opération de recherche est alors en $O(n)$
- La forme la meilleure est celle de l'arbre "équilibré", i.e. où $taille \approx 2^{prof}$ soit $prof \approx \log_2(taille)$

Il faut donc essayer de maintenir l'équilibre des arbres binaires de recherche dans les opérations d'ajout, de suppression etc.

Les arbres AVL

- Premiers arbres binaires équilibrés
- Inventés en 1962 par les russes Adelson-Velsky et Landis (d'où le nom AVL)
- Ces arbres vérifient la propriété suivante :

  La différence entre les hauteurs des fils gauche et des fils droit de tout nœud ne peut excéder 1

Définition des arbres AVL

- Le type des AVL est similaire à celui des arbres binaires
- On ajoute un entier dans les noeuds afin de mémoriser la hauteur des arbres

```
type 'a avl =
   Vide | Noeud of 'a * 'a avl * 'a avl * int
```

On manipule les hauteurs des arbres à l'aide des deux fonctions suivantes :

```
let height = function
  | Vide -> 0
  | Noeud(_, _, _, h) -> h

let creation l v r =
  Noeud(v, l, r, 1 + max (height l) (height r))
```

Ajout d'un élément

```
let rec ajout x = function
  | Vide ->
     creation Vide x Vide
  | Noeud(v, l, r, _) as t ->
     if x = v then t else
     if x < v then
      equilibrage (ajout x l) v r
     else
      equilibrage l v (ajout x r)
```

Équilibrage

Soient deux arbres `l` et `r` équilibrés tels que $|\mathtt{prof}(l) - \mathtt{prof}(g)| \leq 2$

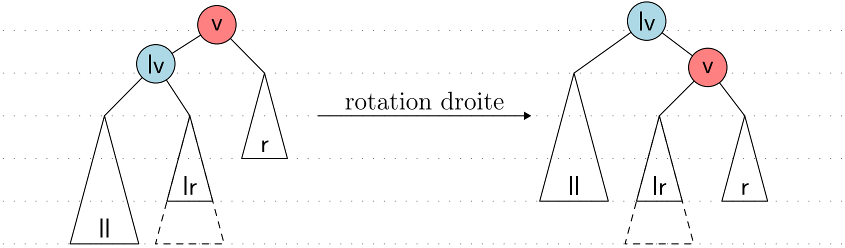
La fonction `equilibrage` construit un arbre équilibré formé des mêmes éléments, et dans le même ordre, que `Noeud(l,v,r)`.

```
let equilibrage l v r =
  let (hl, hr) = (height l, height r) in
  if hl > hr + 1 then begin
   match l with
    | Noeud(lv, ll, lr, _) when height ll >= height lr ->
      creation ll lv (creation lr v r)
    | Noeud(lv, ll, Noeud(lrv, lrl, lrr, _), _)->
      creation (creation ll lv lrl) lrv (creation lrr v r)
    | _ -> assert false
   end else if hr > hl + 1 then begin
     (* cas symétrique *)
  end else creation l v r
```

Équilibrage à droite par simples rotations

Si $\texttt{hl} = \texttt{hr} + 2$ et $\texttt{height}(\texttt{lr}) \leq \texttt{height}(\texttt{ll}) \leq \texttt{height}(\texttt{lr}) + 1$
alors on réalise une rotation simple à droite :

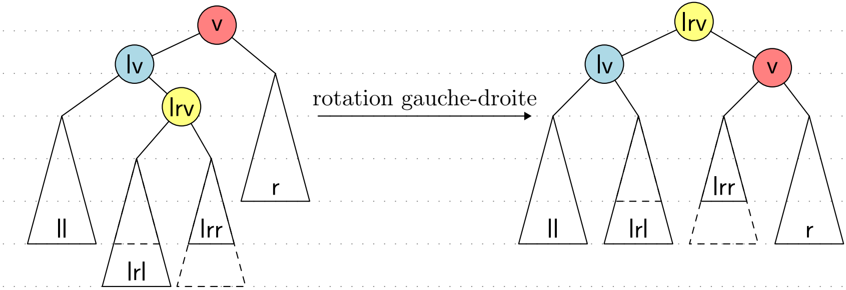
```
creation ll lv (creation lr v r)
```



Équilibrage à droite par doubles rotations

Si $\texttt{hl} = \texttt{hr} + 2$ et $\texttt{height}(\texttt{ll}) = p$ et $\texttt{height}(\texttt{lr}) = p + 1$ alors on réalise une rotation double gauche-droite :

```
creation (creation ll lv lrl) lrv (creation lrr v r)
```



Suppression d'un élément

La suppression d'un élément `x` dans un AVL `Noeud(v, l, r, h)` consiste à :

- Fusionner `l` et `r` si `x`=`v`
- Supprimer `x` dans `l` ou `r` selon la valeur de `x` < `v`
- Re-équilibrer l'arbre obtenu

```
let rec suppression x = function
  | Vide -> Vide
  | Noeud(v, l, r, _) ->
    if x = v then fusion l r else
    if x < v then equilibrage (suppression x l) v r
    else equilibrage l v (suppression x r)
```

Fusion de deux AVLs

Soient deux AVLs `t1` et `t2` tels que $|\texttt{heigth}(t1) - \texttt{height}(t2)| \leq 1$.

- Tous les éléments de `t1` sont plus petits que ceux de `t2`
- La fonction `fusion` construit un AVL formé des mêmes éléments que `t1` et `t2`

```
let fusion t1 t2 =
  match (t1, t2) with
   | (Vide, t) | (t, Vide) -> t
   | (_, _) ->
       equilibrage t1 (min_elt t2) (suppr_min_elt t2)
```

Plus petit élément

La fonction `min_elt` retourne le plus petit élément d'un arbre binaire de recherche.

```
let rec min_elt = function
  | Vide -> raise Not_found
  | Noeud(v, Vide, r, _) -> v
  | Noeud(v, l, r, _) -> min_elt l
```

La fonction `suppr_min_elt` supprime le plus petit élément d'un AVL

```
let rec suppr_min_elt = function
  | Vide -> raise Not_found
  | Noeud(v, Vide, r, _) -> r
  | Noeud(v, l, r, _) ->
     equilibrage (suppr_min_elt l) v r
```