Programmation fonctionnelle - Projet - String Builders

Yoann Boulinguez-Bonnet

12 mai 2022

Table des matières

1. Présentation du sujet

La majorité des langages de programmation fournissent une notion primitive de chaîne de caractères. Si ces chaînes s'avèrent adaptées à la manipulation de mots ou de textes relativement courts, elles deviennent généralement inutilisables sur de très grands textes. L'une des raisons de cette inefficacité est la duplication d'un trop grand nombre de caractères lors des opérations de concaténation ou l'extraction d'une sous-chaîne.

Or il existe des domaines où la manipulation efficace de grandes chaînes de caractères est essentielle (représentation du génome en bio-informatique, éditeurs de texte, ...). Ce projet propose une alternative à la notion usuelle de chaîne de caractères que nous appelons `string_builder`. Un `string_builder` est un arbre binaire, dont les feuilles sont des chaînes de caractères usuelles et dont les noeuds internes représentent des concaténations.

2. Définition du type

Un `string_builder` est donc soit un mot, soit une concaténation de deux autres `string_builder` (noeud).

```
type string_builder =
  | Empty
  | Leaf of int * string
  | Node of int * string_builder * string_builder;;
```

Le code précédent permet d'implémenter totalement les `string_builders` tels que définis dans l'énoncé.

3. Fonctions d'échauffement

3.1. Question 1

La fonction `word` suivante prend en argument une chaîne de caractères et renvoie le `string_builder` constitué d'une seule feuille correspondant.

```
let word x = Leaf (String.length x, x);;
```

On utilise la fonction `String.length` qui renvoie la longueur d'une chaîne de caractères pour définir complètement la feuille.

La fonction `concat` qui prend en argument deux `string_builder` et qui renvoie le nouveau `string_builder` résultant de leur concaténation.

Pour implémenter cette fonction, on utilise la fonction suivante qui renvoie la longueur d'un `string_builder` :

```
let string_builder_length = function
  | Empty -> 0;
  | Leaf(n, x) -> n;
  | Node(n, x1, x2) -> n;;
```

On a donc la fonction `concat` suivante :

```
let concat sb1 sb2 = match (sb1, sb2) with
  | (Empty, sb2) -> sb2;
  | (sb1, Empty) -> sb1;
  | (sb1, sb2) -> Node(string_builder_length sb1 +
                       string_builder_length sb2, sb1, sb2);;
```

3.2. Question 2

On définit une fonction `char_at` qui prend en argument un entier i et un `string_builder` représentant le mot $[c_0, \ldots, c_{n-1}]$, et qui renvoie le caractère ci.

```
let rec char_at i sb = match sb with
  | Empty -> failwith "String builder vide";
  | Leaf(_,x) -> String.get x i;
  | Node(_,sb1,sb2) -> let n1 = string_builder_length sb1 in
      if i < n1 then char_at i sb1
      else char_at (i-n1) sb2;;
```

Si nous n'avons qu'une seule feuille, nous cherchons le $n^{\text{ième}}$ caractère avec la fonction `String.get`.

Dans le cas le plus général, si le nombre n passé en argument est plus petit que la taille du string builder de gauche, on fait la recherche du côté gauche, sinon on cherche du côté droit.

3.3. Question 3

Définir une fonction `sub_string` qui prend en arguments un entier i, `string_builder sb` et un entier m représentant le mot $[c_0, \cdots, c_{n-1}]$ et qui renvoie un `string_builder` représentant le mot $[c_i, \ldots, c_{i+m-1}]$, c'est-à-dire la sous-chaîne de c débutant au caractère i et de longueur m.

```
let rec sub_string i m sb = match sb with
  | Empty -> failwith "String builder vide";
  | Leaf(n, x) -> Leaf(m, String.sub x i m);
  | Node(n, sb1, sb2) when i+m <= string_builder_length sb1
                                -> sub_string i m sb1;
  | Node(n, sb1, sb2) when i >= string_builder_length sb1
                       -> sub_string (i-string_builder_length sb1) m sb2;
  | Node(n, sb1, sb2) -> let n1 = string_builder_length sb1 in
                    concat (sub_string i (n1-i) sb1) (sub_string 0 (m-n1+i) sb2));;
```

Dans le cas d'une feuille, c'est relativement simple : on utilise simplement la fonction `String.sub` et on retourne la feuille correspondante.

Dans le cas d'un noeud, soit le motif recherché est facteur de la partie gauche, soit il est facteur de la partie droite, soit c'est un suffixe de la partie gauche et un préfixe de la partie droite.

Dans ce dernier sous-cas, la seule difficulté est de gérer les indices de début et de fin lors des appels à `sub_string`.

4. Équilibrage

4.1. Question 4

On définit la fonction `cost` qui prend en argument un `string_builder` et qui renvoie son coût selon la définition suivante :

$$\sum_{i=0}^{k} \texttt{length}(sb) \times \texttt{depth}(sb)$$

```
let cost sb =
  let rec aux acc prof s = match s with
    | Empty -> acc;
    | Leaf(n, x) -> aux (acc + n*prof) (prof + 1) Empty;
    | Node(_, s1, s2) -> aux acc (prof+1) s1 + aux acc (prof +1) s2;
  in aux 0 0 sb;;
```

Pour implémenter cette fonction, nous utilisons une fonction auxiliaire qui prend deux accumulateurs en argument : un qui sera retourné et qui représente la somme cherchée et un autre représentant la profondeur.

Dans le cas le plus simple, c'est-à-dire une feuille, on appelle la fonction en incrémentant la profondeur de 1 et l'accumulateur par la longueur de la chaîne de caractère stockée dans la feuille, multipliée par sa profondeur, comme voulu.

Dans le cas le plus général, c'est-à-dire un noeud, on appelle la fonction auxuliaire aux deux sous-`string_builders` gauche et droit, en incrémentant seulement la profondeur de 1.

On appelle cette fonction auxiliaire avec `acc = 0` et `prof = 0` car, à l'état initial, la profondeur est nulle et la somme également.

4.2. Question 5

J'ai choisi de diviser la question 5 en plusieurs étapes.

En premier lieu, je génère un caractère compris entre `a` et `z` en utilisant la fonction `Char.chr`, qui convertit un code ASCII en son caractère correspondant.

```
let rand_chr () = (Char.chr (97 + (Random.int 26)));;
```

En effet, le code ASCII des lettres minuscule est compris entre 97 et 122.

La deuxième étape est de générer une consomme et une voyelle à partir de la fonction précédente. C'est l'objet des deux fonctions suivantes :

```
let rec rand_voy () = let chr = (rand_chr ()) in match chr with
  | 'a' | 'e' | 'i' | 'o' | 'u' | 'y' ->  chr;
  | _ -> rand_voy ();;

let rec rand_con () = let chr = (rand_chr ()) in match chr with
  | 'a' | 'e' | 'i' | 'o' | 'u' | 'y' ->  rand_con ();
  | _ -> chr;;
```

À partir de la fonction précédente, on définit ensuite une fonction qui génère une syllabe aléatoire, en concaténant une consonne et une voyelle :

```
let rec rand_convoy acc syll_number () = match syll_number with
  | 0 -> acc;
  | _ -> rand_convoy (acc ^ (Char.escaped (rand_con ())) ^
                        (Char.escaped(rand_voy()))) (syll_number - 1) ();;
```

La fonction `Char.escaped` convertit un caractère en une chaîne de caractères.

Enfin, on génère un mot aléatoire comportant entre 1 et 3 syllabes :

```
let rand_word () = (rand_convoy "" (1 + (Random.int 2)) ());;
```

Finalement, on génère un `string_builder` aléatoire en utilisant cette dernière fonction. Si la profondeur précisée est de 1, on génère une chaîne aléatoire que l'on retourne dans une feuille correspondante.

Dans le cas où la profondeur précisée est strictement supérieure à 1, on concatène une feuille avec un `random_string` de taille $(i-1)$.

```
let rec random_string i = match i with
  | 0 -> Empty;
  | 1 -> let str = rand_word () in Leaf(String.length(str), str);
  | _ -> let str = random_string (i-1) in Node(i, word (rand_word()), str);;
```

4.3. Question 6

On définit une fonction `list_of_string` qui prend en argument un `string_builder` et qui renvoie la liste des chaînes de caractères dans le même ordre que dans l'arbre (parcours infixe).

```
let rec list_of_string sb = match sb with
  | Empty -> [];
  | Leaf(n,x) -> [x];
  | Node(n,sb1,sb2) -> list_of_string(sb1)@list_of_string(sb2);;
```

On applique récursivement notre fonction sur les deux sous-`string_builder` pour obtenir la liste désirée, sachant que les deux cas de base sont soit la liste vide retournée si l'on tombe sur un `string_builder` vide, soit la liste constituée d'une chaîne de caractère dans le cas d'une feuille.

4.4. Question 7

On résout la question en développant plusieurs fonctions auxiliaires.

— La première étape est de transformer le `string_builder` en une liste de ses feuilles. On utilise la précédente fonction pour obtenir la liste de toutes les chaînes de caractère dans le bon ordre ; liste sur laquelle nous faisons un `map` pour les convertir en feuilles.

```
let list_of_leaves sb = let list = list_of_string sb in
  let f x = Leaf(String.length(x), x) in List.map f list;;
```

— La deuxième étape est de rechercher le coût minimum de concaténation de deux éléments successifs dans la liste. On travaille ainsi avec des références pour parcourir la liste.

Au début, l'indice recherché est égal à 0 et le coût minimum est celui de la concaténation des deux premiers éléments.

On utilise ensuite une fonction auxiliaire pour parcourir la liste entière, qui prend deux arguments dont un entier représentant notre *curseur*. On stocke dans une variable la valeur de la concaténation de l'élément pointé par le curseur et de son successeur. Ensuite, on fait des comparaison entre le coût de concaténation que l'on vient de calculer et le minimum, et l'on échange éventuellement les valeurs.

On utilise enfin `ignore` pour ne pas renvoyer le résultat de la fonction auxiliaire car elle nous renvoie un coût. On renvoie ensuite l'indice minimum en accédant à la valeur de la référence.

```
let min_cost_concat leaves_list =
  let min_index = ref 0 in
  let first_concat = concat (List.nth leaves_list 0) (List.nth leaves_list 1) in
  let min_cost = ref (cost first_concat) in

  let rec aux leaves_list i = match leaves_list with
    | [] -> failwith "Empty list";
    | [sb] -> cost sb;
    | sb1::sb2::q -> let cost_concat = cost (concat sb1 sb2) in
        if cost_concat < !min_cost then (min_cost := cost_concat;
                                         min_index := i;
                                         aux (sb2::q) (i+1);
                                        )
        else aux (sb2::q) (i+1);
  in ignore (aux leaves_list 0);
  !min_index;;
```

— La troisième étape consiste en une fonction remplaçant deux éléments successifs par leur concaténation, connaissant leur indice.

```
let rec replace_concat list i = match list with
  | [] -> failwith "Empty list";
  | [sb] -> failwith "Impossible concatenation because
                      the list contains only one element";
  | sb1::sb2::q -> if i == 0 then let c = concat sb1 sb2 in c::q
                   else let t = replace_concat (sb2::q) (i-1) in sb1::t;;
```

— Enfin, on applique l'algorithme d'équilibrage. On convertit tout d'abord le `string_builder` en une liste de ses feuilles avec la fonction définie à la première étape.
Ensuite, on définit une fonction auxiliaire pour appliquer récursivement notre algorithme tant que nous avons deux éléments dans la liste.
L'algorithme fait en sorte que la taille de la liste diminue à chaque appel à la fonction auxiliaire : lorsqu'il n'y aura plus qu'un argument, il s'agira du `string_builder` équilibré, et donc on le renverra.

```
let balance sb =
  let leaves_list = list_of_leaves sb in
  let rec aux leaves_list = match leaves_list with
    | [] -> failwith "The string builder list is empty."
    | [sb] -> sb
    | leaves_list -> let index_min = min_cost_concat leaves_list in
        let concat_list = replace_concat leaves_list index_min in
        aux concat_list
  in aux leaves_list;;
```

4.5. Question 8

On définit plusieurs fonctions :

— une première `random_list_tree` retourne une liste de i `string_builders` générés aléatoirement ;

— une deuxième, `balanced_random`, crée les `string_builders` équilibrés à partir de la liste initiale ;

— enfin nous implémentons trois fonctions renvoyant le maximum, le minimum et la moyenne des éléments d'une liste.

```
let rec random_list_tree i = match i with
  | 0 -> failwith "Please enter a nonzero value";
  | 1 -> [random_string (Random.int 10)];
  | i -> random_string (Random.int 10)::(random_list_tree (i-1));;

let balanced_random list = List.map balance list;;

let rec max_list l = match l with
  | [] -> failwith "Empty list";
  | [x] -> x;
  | t::q -> max t (max_list q);;

let rec min_list l = match l with
  | [] -> failwith "Empty list";
  | [x] -> x;
  | t::q -> min t (min_list q);;

let rec sum list = match list with
  | [] -> 0
  | h::t -> h + (sum t);;

let mean_list l = let s = sum l in s/(List.length l);;
```

Enfin, nous calculons les gains en coût de la fonction `balance` sur un grand nombre d'arbres générés aléatoirement.

```
let gain i = let random_list = random_list_tree i in
      let random_list_balanced = balanced_random (random_list) in
      let cost_list = List.map cost random_list in
      let cost_list_balanced = List.map cost random_list_balanced in
      let min_cost = min_list cost_list in
      let min_cost_bal = min_list cost_list_balanced in
      let max_cost = max_list cost_list in
      let max_cost_bal = max_list cost_list_balanced in
      let mean_cost = mean_list cost_list in
      let mean_cost_bal = mean_list cost_list_balanced in
      Printf.printf "Min, max and mean cost for the random trees :
                          %d, %d, %d\n" min_cost max_cost mean_cost ;
      Printf.printf "Min, max and mean cost for the balanced random trees :
                          %d, %d, %d\n" min_cost_bal max_cost_bal mean_cost_bal;;
```

5. Description des types

```
val word : string -> string_builder = <fun>
val string_builder_length : string_builder -> int = <fun>
val concat : string_builder -> string_builder -> string_builder = <fun>
val char_at : int -> string_builder -> char = <fun>
val sub_string : int -> int -> string_builder -> string_builder = <fun>
val cost : string_builder -> int = <fun>
val rand_chr : unit -> char = <fun>
val rand_voy : unit -> char = <fun>
val rand_con : unit -> char = <fun>
val rand_convoy : string -> int -> unit -> string = <fun>
val rand_word : unit -> string = <fun>
val random_string : int -> string_builder = <fun>
val list_of_string : string_builder -> string list = <fun>
val list_of_leaves : string_builder -> string_builder list = <fun>
val min_cost_concat : string_builder list -> int = <fun>
val replace_concat : string_builder list -> int -> string_builder list = <fun>
val balance : string_builder -> string_builder = <fun>
val random_list_tree : int -> string_builder list = <fun>
val balanced_random : string_builder list -> string_builder list = <fun>
val max_list : 'a list -> 'a = <fun>
val min_list : 'a list -> 'a = <fun>
val sum : int list -> int = <fun>
val mean_list : int list -> int = <fun>
val gain : int -> unit = <fun>
```

6. Test des fonctions

On utilise les `string_builder` suivant pour réaliser les tests des fonctions :

```
let essai = Node(5,
                 Leaf(1, "H"),
                 Node(4,
                      Leaf(1, "E"),
                      Leaf(3, "LLO")
                     )
                );;

let essai_bis = Leaf(3, "IPF");;
```

Question 1 `word "Vive le Caml"` renvoie bien

```
- : string_builder = Leaf (12, "Vive le caml")
```

`string_builder_length essai` renvoie

```
- : int = 5
```

ce qui est cohérent.

Enfin, lors de l'appel à `concat essai essai_bis`, on obtient :

```
Node (8, Node (5, Leaf (1, "H"), Node (4, Leaf (1, "E"), Leaf (3, "LLO"))),
 Leaf (3, "IPF"))
```

qui est bien un `string_builder` possible résultat de cette concaténation.

Question 2 Lors de l'appel `char_at 3 essai`, on obtient :

```
- : char = 'L'
```

qui est bien le résultat attendu.

Question 3 L'appel `sub_string 2 2 essai`, on obtient également le résultat attendu :

```
- : string_builder = Leaf (2, "LL")
```

Question 4 Le coût du `string_builder` essai est théoriquement 9. Ce résultat est confirmé par la fonction implémentée dans cette question :

```
- : int = 9
```

Question 5 La génération d'un `string_builder` de profondeur 12 renvoie le `string_builder` suivant :

```
- : string_builder =
Node (5, Leaf (4, "time"),
 Node (4, Leaf (4, "lely"),
  Node (3, Leaf (4, "gyno"), Node (2, Leaf (2, "jy"), Leaf (2, "ga")))))
```

Qestion 6 La liste des chaînes de caractères retournée par l'appel à `list_of_strings essai` renvoie bien la valeur attendue :

```
- : string list = ["H"; "E"; "LLO"]
```

Question 7 Le `string_builder` balancé renvoyé par l'appel à `balance essai` est le suivant :

```
- : string_builder =
Node (5, Node (2, Leaf (1, "H"), Leaf (1, "E")), Leaf (3, "LLO"))
```

Question 8 L'appel à la fonction `gain 8` renvoie :

```
Min, max and mean cost for the random trees : 6, 80, 57
Min, max and mean cost for the balanced random trees : 6, 62, 37
```