Rapport Ocaml 2022 : String Builders

KHIDOUR Alae

Mai 2022

1 Introduction : Enoncé du projet

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 Fonctions utilisées

2.1 Question 1

2.1.1 type string_builder

On commence par définir le type *string_builder* :

```
type string_builder =
        |Mot of string * int
        |N of int * string_builder * string_builder
;;
```

2.1.2 word : $string -> string\_builder =< fun >$

word prend en argument une chaîne de caractères et qui renvoie le *string_builder* constitué d'une seule feuille correspondant.

2.1.3 longueur : $string\_builder -> int =< fun >$

longueur renvoie la longueur d'un *string_builder*

2.1.4 concat : $string\_builder -> string\_builder -> string\_builder =< fun >$

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

2.1.5 Tests

```
let w1 = word "Hello "
let w2 = word "world! ";;

let w3 = word "My "
let w4 = word "name ";;
```

```
let w5 = word "is "
let w6 = word "Alae";;

let phrase = concat (concat w1 w2) (concat (concat w3 w4) (concat w5 w6));;
let l = longueur phrase;;
```

2.2 Question 2

2.2.1 char_at : $int- > string\_builder- > char =< fun >$

char_at prend en argument un entier i et un *string_builder* représentant le mot $[c_0; ...; c_{n1}]$, et renvoie le caractère ci. Note : On supposera $0 \leq i < n$, et on utilisera la fonction *String.get* pour obtenir le i ème caractère d'une chaîne de caractères.

-Si le *string_builder* est un Mot alors on revoie le i ème caractère du mot.
-Si le *string_builder* est un Noeud (N) et que l'indice i est dans la partie gauche on rappelle la fonction sur la partie gauche.
-Si le *string_builder* est un Noeud (N) et que l'indice i est dans la partie droite on rappelle la fonction sur la partie droite et avec l'indide $i - longueur\ gauche$.

2.2.2 Tests

```
char_at 0 phrase;;
char_at 1 phrase;;
char_at 6 phrase;;
char_at 8 phrase;;
char_at 16 phrase;;
char_at 20 phrase;;
char_at 26 phrase;;
char_at 27 phrase;;
```

2.3 Question 3

2.3.1 sub_string : $int- > int- > string\_builder- > string\_builder =< fun >$

sub_string qui prend en arguments un entier i, un entier m et un *string_builder* sb représentant le mot $[c_0; ...; c_{n1}]$ et qui renvoie un *string_builder* représentant le mot $[c_i; ...; c_{i+m-1}]$, c'est-à-dire la sous-chaîne de c débutant au caractère i et de longueur m. Note : On supposera $0 \leq i < i+m \leq n$, et on s'attachera à réutiliser dans le *string_builder* résultant autant de sous-arbres de sb que possible.
1) Si le *string_builder* est un Mot, on revoie le Mot à partir du i ème caractère et de longueur l .
2) Si le *string_builder* est un Noeud alors on a deux cas :
i) L'indice est à gauche et la longueur du mot ne déborde pas sur la partie de droite : on rappelle alors la fonction sur la partie de gauche avec le même indice et la même longueur.

ii) L'indice est à gauche et la longueur du mot déborde sur la partie de droite : retourne la concaténation de *sub_string* de sb_left avec comme indice i et comme longueur la longueur de sb_left moins i et *sub_string* de *sb_right* avec comme indice 0 et longueur $m - i$ plus la longueur de sb_left (pour prendre compléter la sous-chaîne avec la longueur nécessaire en partant du début de la chaîne de sb_left).

3) Si l'indice est à droite on rappelle la fonction sur la partie droite et avec l'indide $i - longueur\ gauche$ et la même longueur.

2.3.2 Tests

```
String.sub "Hello" 0 2;;

sub_string 0 3 w1;;
sub_string 0 6 w1;;
```

```
let hw = concat w1 w2;; (* "Hello world! " *)

sub_string 0 3 hw;;
sub_string 0 6 hw;;
sub_string 0 7 hw;;
sub_string 6 6 hw;;
sub_string 0 13 hw;;
sub_string 5 1 hw;;
sub_string 5 3 hw;;

sub_string 5 15 phrase;;
sub_string 5 18 phrase;;
```

2.4 Question 4

2.4.1 cost : $string\_builder -> int =< fun >$

cost qui prend en argument un *string_builder* et qui renvoie son coût selon la définition de l'énoncé.

1) Si le *string_builder* est un Mot on renvoie longueur*profondeur.
2) Si le *string_builder* est un Noeud on renvoie le coût de la partie gauche plus le coût de la partie droite.

2.4.2 Tests

```
cost w1;;
cost hw;;
cost phrase;;
```

2.5 Question 5

2.5.1 rand_chr : $unit -> char =< fun >$

rand_chr renvoie une chaîne de caractère contenant une lettre aléatoire miniscule (de 'a' à 'z').

2.5.2 rand_chaine : $unit -> string =< fun >$

rand_chaine renvoie une chaîne de caractère aléatoire de taille aléatoire entre 1 et 7.

2.5.3 rand_Mot : $unit -> string\_builder =< fun >$

rand_chaine renvoie un Mot aléatoire de taille aléatoire entre 1 et 7.

2.5.4 random_string : $int -> string\_builder =< fun >$

rand_string renvoie un *string_builder* aléatoire de profondeur i et dont les Mot (feuilles) sont de taille aléatoire entre 1 et 7.

Le procédé de création "aléatoire" est le suivant : avec une fonction auxilière on garde en mémoire la profondeur actuelle, puis tant qu'on a pas atteint la profondeur i on rajoute un degré de profondeur de manière équiprobable soit à gauche soit à droite soit dans les 2 côtés et au dernier niveau de profondeur on génèrent un Mot aléatoire avec $rand\_Mot$.

Je pense que ce procédé de création aléatoire est améliorable car ce procédé donne de manière équiprobable des *string_builder* de la forme $N(Mot, droite)$, $N(gauche, Mot)$ ou $N(gauche, droite)$ avec *gauche* et *droite* des *string_builder* $\neq Mot('...', l)$ de part la toute première itération et choix de forme pour le *string_builder*, alors qu'en réalité, avec un simple raisonnement de dénombrement, on peut montrer qu'il y a beaucoup plus d'arbres de la forme $N(gauche, droite)$ que $N(gauche, Mot)$ ou $N(Mot, droite)$. Il faudrait donc pondérer les 3 choix à chaque itération par le nombre d'arbres de forme $N(Mot, droite)$, $N(gauche, Mot)$ ou $N(gauche, droite)$ sur le nombre total d'arbres possible (Ce qui augmente considérablement la difficulté).

2.5.5 Tests

```
rand_Mot ();;
random_string 3;;
```

2.6 Question 6

2.6.1 list_of_string : $string\_builder -> string\ list =< fun >$

list_of_string prend en argument un *string_builder* et renvoie la liste des chaînes de caractères dans le même ordre que dans l'arbre (parcours infixe).

2.6.2 Tests

```
list_of_string phrase;;
list_of_string (random_string 10);;
```

2.7 Question 7

2.7.1 list_of_Mot : $string\_builder -> string\_builder\ list =< fun >$

list_of_Mot prend un *string_builder* et le renvoie sous forme de liste des Mots qui le composent.

2.7.2 concat_i_next : $string\_builder\ list -> int -> string\_builder\ list =< fun >$

concat_i_next prends une liste de *string_builder* et un indice i et concatène le $i$ème *string_builder* et le $(i+1)$ème *string_builder* et renvoie une nouvelle liste de *string_builder* de taille inférieure (-1).

2.7.3 min : $'a -> 'a -> 'a =< fun >$

min Renvoie le min de deux nombres.

2.7.4 max : $'a -> 'a -> 'a =< fun >$

max Renvoie le max de deux nombres.

2.7.5 indice_min_cost : $string\_builder\ list -> int =< fun >$

indice_min_cost Renvoie l'indice de la concaténation de deux éléments successifs de plus faible coût d'une liste de *string_builder*. Pour cela on utilise un fonction *aux* qui prend en paramètre une liste, un minimum (actuel) et deux entiers qui sont *ind_act* et *ind_min_act* qui sont respectivement l'indice actuel et l'indice actuel du coût de concaténation le plus faible. Elle va ensuite parcourir la liste en vérifiant si le coût de concaténation de deux éléments est plus faible que le *min_act* : si oui elle met à jour le *min_act* et l'entier *ind_min_act* qui stocke la valeur que l'on va retourner.

2.7.6 balance : $string\_builder -> string\_builder =< fun >$

balance prend un *string_builder* et renvoie le *string_builder* équilibrer correspondant selon l'algo du sujet.

2.7.7 Tests

```
list_of_Mot phrase;;

let l = "ab"::"abc"::"abcd"::"e"::"a"::"er"::[];;
let l2 = List.map word l;;

let phrase2 = concat w1 (concat w2 (concat w3 (concat w4 (concat w5 w6) ) ) );;

balance phrase2;;
```

```
let lp = list_of_Mot phrase2;;
let i1 = indice_min_cost (lp);;
let etape1 = concat_i_next lp i1;;
let i2 = indice_min_cost (etape1);;
let etape2 = concat_i_next etape1 i2;;
let i3 = indice_min_cost (etape2);;
let etape3 = concat_i_next etape2 i3;;
let i4 = indice_min_cost (etape3);;
let etape4 = concat_i_next etape3 i4;;
let i5 = indice_min_cost (etape4);;
let etape5 = concat_i_next etape4 i5;;

cost phrase;;
cost (balance phrase);;

cost phrase2;;
balance phrase2;;
cost (balance phrase2);;
```

2.8 Question 8

2.8.1 type stats

Le type stats servira "d'étude sur un échantillon aléatoire" de liste de *string_builder*, à partir de cette liste on va calculer la liste des coûts puis son min, max, moyenne et mediane puis on va calculer la liste des mêmes *string_builder* équilibrés puis refaire la même chose pour comparer. On donne aussi la liste des gains terme à terme et le gain total qui n'est que la somme de la liste des gains.

```
type stats =
        { list_sb : string_builder list ;
          list_cost : int list;
          minimum : int ;
          maximum : int ;
          moyenne : float ;
          mediane : float ;
          list_sb_balanced : string_builder list ;
          list_cost_balanced : int list;
          minimum_balanced : int ;
          maximum_balanced : int ;
          moyenne_balanced : float ;
          mediane_balanced : float ;
          list_gain : int list;
          gain_tot : int;
        }
;;
```

2.8.2 generate_random_sb_list : $int -> string\_builder\ list$

$generate_random\_sb\_list$ génére aléatoirement une liste de taille donnée contenant des *string_builder* de profondeurs aléatoires entre 1 et 9 (choix arbitraires car plus que 9 ne fait que rallonger le temps de calcul).

2.8.3 list_of_cost : $string\_builder\ list -> int\ list =< fun >$

$list\_of\_cost$ prend une liste de *string_builder* et renvoie la liste des coûts respectifs des *string_builder*.

2.8.4 min_list : $'a\ list -> 'a =< fun >$

min_list renvoie le min d'une liste.

2.8.5 max_list : $'a\ list- > '\ a =< fun >$

max_list renvoie le max d'une liste.

2.8.6 somme_list : $int\ list- > int =< fun >$

somme_list renvoie la somme des éléments d'une liste.

2.8.7 moyenne_list : $int\ list- > float =< fun >$

moyenne_list renvoie la moyenne des éléments dúne liste.

2.8.8 mediane_list : $int\ list- > float =< fun >$

médiane_list renvoie la médiane des éléments dúne liste.

2.8.9 diff_deux_list : $int\ list- > int\ list- > int\ list =< fun >$

diff_deux_list prends l1 et l2 (de même taille) et renvoie la liste des différences terme à terme (l1-l2).

2.8.10 generate_stat : $unit- > stats =< fun >$

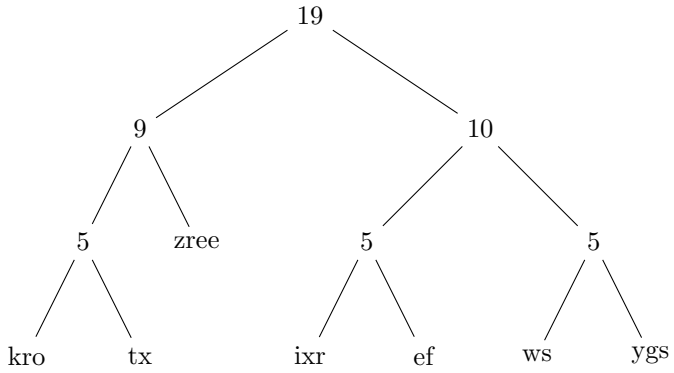
generate_stat génére un élément de type *stats* contenant une liste de *string_builder* de taille 300 (échantillon de taille 300) sur laquelle il se base pour calculer tous les attributs du type stats.

2.8.11 Tests

```
let test =(1::2::3::4::3::5::[]);;

min_list test;;

max_list test;;

somme_list test;;

moyenne_list test;;

mediane_list test;;



let test = generate_stat ();;

test.minimum;;
test.minimum_balanced;;

test.maximum;;
test.maximum_balanced;;

test.moyenne;;
test.moyenne_balanced;;

test.mediane;;
test.mediane_balanced;;

test.list_gain;;
test.gain_tot;;
```

2.9 Conclusion

L'implémentation du type *string_builder* ainsi que des différentes fonctions permet de saisir et l'intêret des arbres équilibrés.
En observant les valeurs obtenues par la fonction *generate_stat*, on se rend compte que la fonction balance offre un réel gain en terme de coût pour les *string_builder*.
Cependant, en analysant visuellement la liste des gains (attribut du type *stats*) j'ai remarqué l'apparition de quelques valeurs negatives, que je n'ai pas compris au début car je pensais que l'arbre équilibré avait toujours un coût inférieur ou égale à l'abre d'origine. Mais en fait il existe des contres exemple à ceci comme l'arbre suivant (qui a un coût = 53) :



En l'équilibrant selon l'algo donné,infomatiquement mais j'ai aussi fait la même chose manuellement on obtient l'arbre suivant(qui a un coût = 54>53) :