Projet d'IPFL

Hélène de Foras du Bourgneuf

Avril 2022

1 Introduction

Le but du sujet est de créer un nouveau type `string_builder` pour manipuler efficacement des grandes chaînes de caractères. Un `string_builder` est un arbre binaire dont la définition du type est ci-dessous.

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

Un `string_builder` est non vide, il contient soit une `Leaf` composée d'une `string` et de sa longueur, soit un `Node` composé de deux `string_builder` et du nombre total de caractères qu'ils contiennent.

2 Description des fonctions et des choix faits

2.1 Echauffement

2.1.1 Question 1

```
val word : string -> string_builder = <fun>
```

La fonction `word` prend en paramètre une `string` et la transforme en `Leaf`. Elle utilise la fonction `length` de la librairie `String`.

```
val concat : string_builder -> string_builder ->
string_builder = <fun>
```

La fonction `concat` a deux `string_builder` en paramètres, et renvoie la
concaténation des deux. Pour cela, on créé un `Node`, et ses deux fils sont les paramètres rentrés.

2.1.2 Question 2

```
val char_at : int -> string_builder -> char = <fun>
```

La fonction récursive `char_at` parcourt l'arbre passé en paramètre en incrémentant `i` jusqu'à trouver le i^{eme} caractère du `string_builder`.

2.1.3 Question 3

```
val sub_string : int -> int -> string_builder -> string_builder = <fun>
```

On utilise la fonction `sub` de la librairie `String` qui, à partir d'une string, renvoie une string de la longueur voulue commençant à la position demandée. On traite les trois cas :

```
Leaf(a,x)
Node(Leaf(a,x),b,y)
Node(Node(a,b,x),c,y)
```

2.2 Equilibrage

2.2.1 Question 4

```
val cost : string_builder -> int = <fun>
```

La fonction `cost` fait appel à une fonction auxiliaire récursive, qui somme le produit de la profondeur par la longueur de la `string`.

2.2.2 Question 5

Pour vérifier que la profondeur de l'arbre aléatoire correspondait à ce que je voulais, j'ai écrit la fonction `depth`, qui comme son nom l'indique, renvoie la profondeur du `string_builder` passé en paramètre.

```
val depth : string_builder -> int = <fun>
```

Les feuilles ont 0 de profondeur, seuls les noueds font augmenter la profondeur.

```
val random_string : int -> string_builder = <fun>
```

Cette fonction utilise la librairie `Random`. Elle renvoie un arbre dont la profondeur a été passé en paramètre. La fonction `random_string` est récursive et fait appel à deux fonctions auxiliaires, l'une renvoie la longueur de la `string` correspondant à un arbre, l'autre renvoie un mot aléatoire. L'arbre renvoyé est binaire complet. Les chaînes de caractères de ses feuilles contiennent entre 1 et 10 caractères de code ASCII entre 65 et 122.

Par exemple:

```
val sb_test : string_builder =
  Node
   (Node
     (Node (Node (Leaf ("RriBo", 5), Leaf ("tlwmx", 5), 10),
       Node (Leaf ("R", 1), Leaf ("ZN", 2), 3), 13),
     Node (Node (Leaf ("XUIOn", 5), Leaf ("A", 1), 6),
      Node (Leaf ("[Gu", 3), Leaf ("hhp", 3), 6), 12),
     25),
    Node
     (Node (Node (Leaf ("eb", 2), Leaf ("v", 1), 3),
       Node (Leaf ("p", 1), Leaf ("^", 1), 2), 5),
     Node (Node (Leaf ("w", 1), Leaf ("tx", 2), 3),
      Node (Leaf ("_", 1), Leaf ("gf", 2), 3), 6),
     11),
    36)
```

Cet arbre est de profondeur 4.

2.2.3 Question 6

```
val list_of_string : string_builder -> string list = <fun>
```

Cette fonction fait un parcours infixe du `string_builder` et renvoie une `string list` des chaînes de caractères qu'il contient.

2.2.4 Question 7

```
val balance : string_builder -> string_builder = <fun>
```

La fonction `balance` est en trois parties :

1. `weak_cost` renvoie le coût minimal de concaténation de deux `string_builder` consécutifs de la liste
2. `retire_concat` renvoie la liste passée en paramètre, mais avec la concaténation des deux `string_builder` de plus faible coût à leur place
3. `aux_b` applique la fonction précédente jusqu'à ce que la liste ne contienne plus qu'un seul `string_builder`, et le renvoie tel quel

Par exemple :

```
# let sb_test=random_string 3;;
val sb_test : string_builder =
  Node
   (Node (Node (Leaf ("Ez", 2), Leaf ("zz", 2), 4),
     Node (Leaf ("r[fJoEC", 7), Leaf ("mTqhDTZX‘", 9), 16), 20),
    Node (Node (Leaf ("LthI", 4), Leaf ("]a", 2), 6),
     Node (Leaf ("xIWEUz", 6), Leaf ("ycY\\i_n[K_", 10), 16), 22),
    42)
# let balance_test=balance sb_test;;
val balance_test : string_builder =
  Node
   (Node
     (Node (Node (Leaf ("Ez", 2), Leaf ("zz", 2), 4), Leaf ("r[fJoEC", 7),
       11),
      Node (Leaf ("mTqhDTZX‘", 9), Node (Leaf ("LthI", 4), Leaf ("]a", 2), 6),
       15),
      26),
    Node (Leaf ("xIWEUz", 6), Leaf ("ycY\\i_n[K_", 10), 16), 42)
# let cost_t=cost sb_test;;
val cost_t : int = 126
# let cost_b=cost balance_test;;
val cost_b : int = 120
```

2.2.5 Question 8

```
val gain : int -> int -> int * float * int * int = <fun>
```

La fonction récursive `gain` créé le nombre d'arbres aléatoires demandés de la profondeur passée en paramètre, puis elle calcule la différence de coûts avec `balance` et rajoute cette valeur à la liste. Enfin, elle fait appel à la fonction `values` qui renvoie quatres valeurs : le minimum, la moyenne, la médiane et le maximum des coûts des arbres alétoires par rapport à la fonction `balance`.

```
val values : int list -> int * float * int * int = <fun>
```

3 Cas de tests

```
# let () = assert (word "Hello" = Leaf("Hello",5));;
# let sb = Node(Node(Leaf("Co",2),Leaf("u",1),3),Leaf("cou",3),6);;
val sb : string_builder =
  Node (Node (Leaf ("Co", 2), Leaf ("u", 1), 3), Leaf ("cou", 3), 6)
# let sb2 = Node(Leaf(" le ",4), Node(Node(Leaf("cha",3),Leaf("m",1),4),
  Node(Leaf("e",1),Leaf("au",2),3),7),11);;
val sb2 : string_builder =
  Node (Leaf (" le ", 4),
   Node (Node (Leaf ("cha", 3), Leaf ("m", 1), 4),
    Node (Leaf ("e", 1), Leaf ("au", 2), 3), 7),
   11)
# let () = assert (concat sb sb2=Node(sb,sb2,17));;
# let () = assert (char_at 4 sb2 ='c');;
# let () = assert (sub_string 1 4 sb =
  Node(Node(Leaf("o",1),Leaf("u",1),2),Leaf("co",2),4));;
# let () = assert (cost sb = 9);;
# let () = assert (depth sb = 2);;
# let () = assert (depth (random_string 2) = 2);;
# let () = assert (list_of_string sb =["Co";"u";"cou"]);;
# let () = assert (balance sb2 = Node(Node(Leaf(" le ",4),Leaf("cha",3),7),
```

```
# let (a,b,c,d) = (gain 15 3);;
val a : int = -7
val b : float = 1.4666666666666663
val c : int = 0
val d : int = 7
# let () = assert (a<=c);;
# let () = assert (b<= float_of_int d);;
# print_string ("ok ! \n");;
ok !
- : unit = ()
# print_string ("ok ! \n");;
ok !
- : unit = ()
```

4 Conclusion

On remarque que en moyenne, la fonction `balance` réduit le coût des `string_builder`. Dans les cas de tests, elle diminue de 1.5 le coût d'arbres de profondeur 3.

Plus les chaînes de caractères sont longues, plus cette fonction est efficace. Pour des chaînes de moins de 30 caractères (15 caractères en moyenne), le gain en coût est d'environ 7, et pour des chaînes de moins de 50 caractères, le gain est d'environ 9 !