Rapport de projet IPF 2022

14 mai 2022

Table des matières

Résumé

Rapport de projet de programmation fonctionnelle de 2022 [1]

1 Présentation du projet

1.1 But 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.

1. Page web : https://web4.ensiie.fr/~stefania.dumbrava/IPF2022/ipfl_projet_2022.pdf

1.2 Principe

L'intérêt des string builder est d'offrir une concaténation immédiate et un partage possible de caractères entre plusieurs chaınes, au prix d'un accès aux caractères un peu plus coûteux. Un string builder est donc soit un mot (feuille), soit une concaténation de deux autres string builder (noeud). Pour des raisons d'efficacité, on conserve dans les feuilles aussi bien que dans les noeuds la longueur length(c) de la chaıne de caractères c correspondante

Nous allons utiliser la structure d'arbre binaire de recherche et utiliser des fonctions appelant les modules List, Random et String pour créer les fonctions qui parcoureront ces string builder.

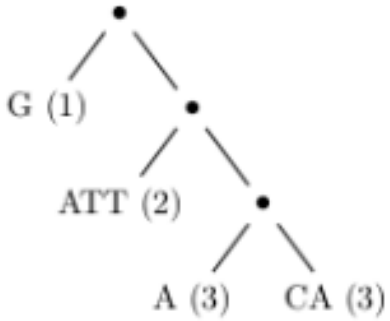

FIGURE 1 – Un string builder

Le string builder ci dessous représente le mot GATTACA obtenu par concaténations des 4 chaînes ["G","ATT","A","CA"]

Voici une définition possible :

```
type string_builder = Feuille of int*string | Noeud of int*string_builder*string
```

le type int permettant de pouvoir créer une fonction

```
val length : string_builder -> int
```

qui stockera la longueur du type nouvellement crée

1.3 La gestion des fonctions

Voici le code de l'interface gérant les fonctions principales :

```
type string_builder

val word : string -> string_builder
```

```
val concat : string_builder -> string_builder -> string_builder

val char_at : int -> string_builder -> char

val sub_string : int -> int -> string_builder -> string_builder

val cost : string_builder -> int

val random_string : int -> string_builder

val list_of_string : string_builder -> string list

val balance : string_builder -> string_builder

val analyse : int -> int * int * float * float
```

Pour la fonction random string, il a fallu implémenter un certain nombre de fonctions intermédiaire manipulant le code ASCII des caractères et utilisant le module Random, afin de pouvoir générer aléatoirement un string builder

```
val int_to_char : int -> char

val randTaille : unit -> int

val randChar : unit -> string

val randStr : unit -> string

val leftOrRight : unit -> int
```

De même, pour équilibrer le poids des feuilles, il a fallu découper la recherche du coût minimal en plusieurs fonctions intermédiaires afin d'appliquer l'algorithme du parcours infixe (feuille gauche, noeud racine, feuille droite) ; ce découpage a du aussi se faire en utilisant un type intermédiaire string builder list.

```
val getFlist : string_builder -> string_builder list

val cosAfterCon : string_builder -> string_builder -> int

val getCostMin1 : int -> string_builder list -> int

val getCostFirst : string_builder list -> int

val getCostFirst : string_builder list -> int
```

2 Gestion du projet

L'implémentation des fonctions sub string et balance ont été les plus difficiles, l'une pour pour comprendre l'algorithme, l'autre pour la découper le travail en utilisant le fonction list of string.

Pour construire le substring entre $[c_i, ..., ci + m - l]$:
- si i+m<=l on garde le string builder gauche - si i>=l on garde le string builder droit - si l-m<i<l on concatène récursivement les 2 sur les feuilles de chacun grâce à la fonction concat

Pour générer un string builder aléatoire et implémenter random string, il a d'abord fallu générer un chiffre aléatoire, l'associer à un caractère ASCII, vérifier que le code ASCII est compris entre 65 et 122 pour obtenir des lettres afin de former des mots puis concaténer avec String.concat la liste de caractères nouvellement obtenus

Pour construire les fonctions intermédiaires afin d'obtenir balance, il a aussi fallu découper la tâche selon l'algorithme de la question 7 en découpant d'abord le string builder en une liste de string, puis en utilisant la concaténation naturelle entre 2 feuilles (@) pour calculer le coût d'accès dans cette liste, puis trier pour calculer le coût le plus faible

Enfin pour la question 8, j'ai écrit les fonctions moyenne avec un compteur et mediane en me servant de la parité du nombre d'éléments grâce à List.nth

2.1 Assertions / tests

```
let exemple = Noeud(7,Feuille(1,"G"),Noeud(6,Feuille(3,"ATT"),Noeud(3,Feuille(1,

let cas = Noeud(6,Noeud(3,Feuille(1,"A"),Noeud(2,Feuille(1,"A"),Feuille(1,"A")))

let test = Noeud (13,Noeud (7, Feuille (1, "G"),Noeud (6, Feuille (3, "ATT"),Noe
Noeud (2, Feuille (1, "A"), Feuille (1, "A"))),Feuille (3, "ATT")));;

let () = assert ( length exemple = 7 );;

let () = assert ( word "exemple" = Feuille(7,"exemple"));;

let () = assert ( concat exemple cas = test);;

let () = assert ( char_at 3 exemple = 'T');;

let () = assert ( sub_string 1 2 exemple = Feuille (2, "AT"));;
```

```
let () = assert ( cost exemple = 16);;

let () = assert ( random_string 4 != random_string 4);;

let () = assert ( list_of_string exemple = ["G"; "ATT"; "A"; "CA"]);;

(* cas non optimal pour balance*)
let () = assert ( balance exemple = Noeud(7,Noeud(4,Feuille(1,"G"),Feuille(3,"AT

let () = assert ( List.length( echantillon 10) = 10);;
let () = assert ( echantillon 10 != echantillon 10);;

let l = [1; 3; 5; 7; 9];;

(* ne marche pas pour les n gatifs*)
let () = assert ( min_number l = 1);;
let () = assert ( max_number l = 9);;
let () = assert ( moyenne l = 5.);;
let () = assert ( mediane l = 5);;
```

2.2 Bonus : implémentation des graphes des string builder

J' ai penser à générer des images au format png grâce à la commande : dot -Tpng fichier.dot > fichier.png

Le temps m'a manqué pour automatiser le processus à chaque création de string builder dans le code

3 Conclusion

Des améliotations peuvent être apportés à mon projet , notamment :

- Optimiser l'analyse des coûts en optant pour un autre tri
- Créer un Makefile fonctionnel gérant automatiquement les dépendances (ocamlbuild)
- Créer une fonction permettant de générer les arbres au format dot