Extrait agrégation d'informatique 2022 sujet 0
Programmation fonctionnelle

Dans ce problème, on utilise le langage OCaml. On se donne le type `tree` suivant pour représenter des arbres binaires :

```
type tree = E | N of tree * tree
```

Le constructeur `E` représente l'arbre vide et le constructeur `N` représente un nœud, avec ses deux sous-arbres. Ici, les nœuds ne contiennent pas d'information, car seule la forme des arbres nous intéresse. La hauteur d'un arbre t, notée $h(t)$, est définie par

$$\begin{aligned} h(\texttt{E}) &= 0, \\ h(\texttt{N}(l,r)) &= 1 + \max(h(l), h(r)). \end{aligned}$$

Elle peut être calculée par la fonction `height` suivante, de type `tree -> int` :

```
let rec height t =
  match t with
  | E        -> 0
  | N (l, r) -> 1 + max (height l) (height r)
```

Question 1. Indiquer le nombre *total* d'appels à la fonction `height` dans le calcul de `height (N (N (E, N (E, E)), N (E, E)))`.

Question 2. On se donne la fonction suivante :

```
let rec left t n =
  if n = 0 then t else left (N (t, E)) (n - 1)
```

Donner son type et expliquer précisément ce que fait cette fonction.

Question 3. On cherche maintenant à exécuter les deux lignes de code suivantes :

```
let t = left E 1000000
let h = height t
```

La première ligne est exécutée sans problème mais la seconde provoque l'erreur suivante :

```
Fatal error: exception Stack_overflow
```

Expliquer cette erreur.

Programmation par continuation. Pour parvenir à calculer la hauteur en toute circonstance, une solution consiste à adopter un style de programmation dit *par continuation*. Plutôt que de calculer directement la hauteur $h(t)$ d'un arbre t, on va calculer $k(h(t))$ pour une fonction k quelconque. La hauteur s'en déduira alors en prenant pour k la fonction identité. La figure 1 contient

```
let rec aux1 t k =
  match t with
  | E ->
      k 0
  | N (l, r) ->
      aux1 l ((*1*) fun hl ->
        aux1 r ((*2*) fun hr ->
          k (1 + max hl hr)))

let height1 t =
  aux1 t ((*3*) fun h -> h)
```

FIGURE 1 – Une autre façon de calculer la hauteur.

un code OCaml qui réalise cette idée. On prendra le temps de bien lire et de bien comprendre ce code, en prêtant notamment attention au parenthésage. On note qu'il y a cinq fonctions en jeu : les deux fonctions `aux1` et `height1` et les trois fonctions anonymes respectivement marquées `(*1*)`, `(*2*)` et `(*3*)`.

Question 4. Indiquer quelles sont *toutes* les fonctions successivement appelées pendant le calcul de `height1 (N (E, N (E, E)))`. On indiquera les appels à `aux1` mais aussi les appels aux fonctions marquées `(*1*)`, `(*2*)` et `(*3*)`. Il n'est pas demandé d'indiquer les paramètres passés à ces différents appels, mais seulement la séquence des appels.

Question 5. Donner le type de la fonction `aux1`.

Question 6. Montrer que la fonction `height1` calcule bien la hauteur.

Question 7. On définit la taille d'un arbre t, notée $|t|$, comme son nombre de nœuds, c'est-à-dire

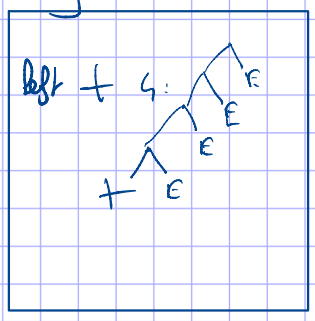
$$\begin{aligned} |\texttt{E}| &= 0, \\ |\texttt{N}(l,r)| &= 1 + |l| + |r|. \end{aligned}$$

Montrer que le calcul de `height1 t` est en $O(|\texttt{t}|)$. Indication : Montrer que la complexité de `aux1 t k` est bornée par $\alpha|\texttt{t}| + \beta + |\texttt{k}|$ où α et β sont deux constantes que l'on déterminera et où $|\texttt{k}|$ désigne le coût de la fonction `k`. Toute séquence bornée de constructions OCaml atomiques pourra être considérée de coût constant.

Défonctionnalisation. On peut modifier le code de la figure 1 pour qu'il n'utilise plus de fonctions anonymes, mais des valeurs d'un type somme OCaml qui représente les différentes fonctions en jeu. On appelle cela la *défonctionnalisation*. La figure 2 contient un squelette de code OCaml qui réalise cette idée. Le type `cont` est le type somme qui représente les trois fonctions anonymes `(*1*)`, `(*2*)` et `(*3*)`. La fonction `apply` permet d'appliquer une continuation `k` de type `cont` à une valeur `v` de type `int`. La fonction `aux2` est l'analogue de la fonction `aux1`, avec `t` de type `tree` et `k` de type `cont`.

Question 1): 8 appels à la fonction Height

Question 2)



Left + n : tree -> int -> tree

Cette fonction crée un fils à droite à chaque appel.

Question 3) La fonction Height ne terminant pas par un appel à elle-même, on a pas de recursivité terminale, ce qui provoque un stack_overflow après trop d'appels.

Question 4):

Height 1 -> aux 1 -> aux 1 -> aux 1

Question 5:

aux 1: tree -> (int -> int) -> int

```
type cont =
  | K1 of tree * cont
  | K2 of int * cont
  | K3

let rec aux2 t k =
  match t with
  | E        -> apply ...
  | N (l, r) -> aux2 ...

and apply k v =
  match k with
  | K1 (r, k) -> aux2 ...
  | K2 (h, k) -> apply ...
  | K3        -> ...

let height2 t =
  aux2 ...
```

FIGURE 2 – Une troisième façon de calculer la hauteur.

Question 8. Compléter le code des fonctions `aux2`, `apply` et `height2`, en donnant les six morceaux de code aux endroits marqués par points de suspension. Le code inséré ne doit contenir *aucun* appel de fonction.

Question 9. L'un des intérêts de la défonctionnalisation est son application à des langages qui ne supportent pas les fonctions anonymes. Discuter de la réalisation du programme de la figure 2 dans le langage C. On ne demande pas d'écrire tout le code, mais seulement les types et les profils des fonctions.

Application au tri fusion. La figure 3 contient le code OCaml d'un tri fusion, sous la forme d'une fonction `mergesort` qui prend en paramètre une liste `l` et renvoie une nouvelle liste, triée par ordre croissant, contenant les mêmes éléments. Deux fonctions auxiliaires sont utilisées : la fonction `split` découpe une liste en deux listes de même longueur (à un près) et la fonction `merge` fusionne deux listes supposées déjà triées par ordre croissant.

Question 10. Tel que le code de la figure 3 est écrit, il est susceptible de provoquer un débordement de pile, tant dans la fonction `split` que dans la fonction `merge`. Réécrire ce code dans un style par continuation (dans le style de la figure 1 et non de la figure 2), en ajoutant une fonction `k` en paramètre des fonctions `split` et `merge`.

* *
*

```
let rec split l =
  match l with
  | []     -> [], []
  | x :: r -> let l1, l2 = split r in l2, x :: l1

let rec merge l1 l2 =
  match l1, l2 with
  | [], l | l, [] ->
      l
  | x1 :: r1, x2 :: r2 ->
      if x1 <= x2 then x1 :: merge r1 l2 else x2 :: merge l1 r2

let rec mergesort l =
  match l with
  | [] | [_] ->
      l
  | _ ->
      let l1, l2 = split l in merge (mergesort l1) (mergesort l2)
```

FIGURE 3 – Tri fusion d'une liste.