Projet individuel d'introduction à la programmation fonctionnelle : Groupe 4

RAHMANY AFIZULLAH

Introduction

Ce projet a pour but d'implémenter d'implémenter un petit langage de programmation manipulant des entiers et des variables.
Nous représenterons les expressions de notre langage par un arbre dont les feuilles sont soit des entiers soit des variables et dont le nœuds sont des sommes ou des produits.

Question 1)

J'ai choisi d'utiliser un filtre sur le type expr, les cas terminales étant le constructeur Var of String où il suffit de renvoyer la chaîne de caractère et le cas Int of int où il suffit d'utiliser int_of_string.

```
@type : expr_to_string : expr -> string
@example : let x = Plus (Int 5, Int 6);;
           let y = expr_to_string x;;
           val y : string = "(5+6)"
```

Question 2)

J'utilise une fonction récursive avec un filtre pour trouver les variables dans les sommes et les produits. La liste garde l'ordre naturel dans lequel les variables se trouvait dans l'expression en argument.

```
@type : variables : expr -> string list
@type : expr_to_string : expr -> string
@example : let x = Mult (Var "h", Int 6);;
           let y = variables x;;
           val y : string list = ["h"]
```

Question 3)

J'utilise une fonction récursive avec un filtre pour simplifier le contenu des sommes et des produits.

```
@type : expr -> int
@param : expr
@return : int (= expr sous forme simplifié)
@example : simplify (Plus(Int 5, Int 6));;
           - : int = 11
```

Question 4)

J'utilise la fonction variables précédente pour avoir la liste des variable et il ne me reste plus qu'à compter les occurrences de la variable que je veux.

```
@type : : expr -> string -> int
@example :# x;;
- : expr = Mult (Var "h", Int 6)
# y;;
- : string list = ["h"]
# nbe_occurences x "h";;
- : int = 1
```

Question 5)

J’utilise une fonction récursive avec un filtre, les cas de bases sont les variables où je substitue la variable si nécessaire et les entiers où je laisse en l’état. Puis j’applique la fonction sur les opérandes des sommes et des produits.

@type : expr -> string -> string -> expr
@param : e (= expr), x (= string, nom de variable), v (= expr)
@return : expr (= l'expression e où on l'on a substitué les x en v)
@example : # let x = Plus (Plus (Var "e", Int 5), Var "k")
substitute x "e" "f";;
- : expr = Plus (Plus (Var "f", Int 5), Var "k")

Question 6)

Dans les fonctions suivant j’utiliserai x et y que j’ai déclaré en dessous comme exemples :

```
let y = If (Int 0, Plus (Int 5, Int 6), Var "b");;
let z = If (Var "bleu", Plus (Int 5, Int 6), Var "b")
```

Ils suffisaient de modifier toutes les fonctions précédentes en ajoutant le cas de la conditionnelle prenant la forme : (t= cas true, f= cas false)

```
|If (e,t,f) -> if e = Int 0 then (fonction f x v) else (fonction t x v)
```

Question 7)

On ajoute un lieur de la forme Let nom_variable = expression1 in expression2 qui se traduit par le constructeur Soit (nom_variable, expression1, expression2)
On introduit la notion d’environnement qui permet de lier des nom de variables à des expressions, en fait le constructeur Soit permet de créer des variables locales, l’environnement est juste la réunion de ces variables locales. Il faut donc arriver à simplifier une expression en tenant compte de son environnement.

<u>Cas d’une stratégie d’appel par valeur :</u>

j’utilise une fonction auxiliaire substitution_val qui remplit le même rôle que la fonction substitute précédente mais en ajoutant le cas du type soit.

@type : substitute_val : string * expr -> expr -> expr
@param : un couple (x,v) avec x nom de variable et v l'expression associé et une expression e
@return : l'expression e où les occurrences de x ont été remplacé par l'expression v dans e

```
let rec substitute_val (x,v) e = match e with
...
```

|Soit (h,i,j) -> substitute_val (x,v) (substitute_val (h,i) j))

dans le cas d'un Soit (h,i,j) (= e, l’expression à évaluer, on veut connaître l’expression simplifié de j finalement) on évalue d'abord j dans (h,i) puis dans (x,v) donc on évalue j dans (x,v) augmenté de (h,i) il s’agit donc d’un appel par valeur.
Il ne reste plus qu’à appliquer cette fonction à tous les éléments de la liste décrivant l’environnement.

@type : eval_nom : expr -> (string * expr) list -> expr

<u>Cas d’une stratégie d’appel par nom :</u>

j’utilise une fonction auxiliaire similaire à la précédente mais qui interdit d’évaluer autre chose qu’un cas de base qui est de constructeur autre que Soit. Pour le cas où on tombe sur un Soit je fais un failwith pour être sur de ne pas tricher.

Ensuite dans ma fonction eval_nom, dans le cas d’un Soit :

@type : eval_nom : expr -> (string * expr) list -> expr

```
let rec eval_nom e env = match env with
|[] -> e
|a::r -> match a with
        |(m, Soit (v,w,u)) -> eval_nom (substitute_nom (m,u) e) ((v,w)::r)
        | _ -> eval_nom (substitute_val a e) r
```

j’augmente l’environnement du couple (v,w) et je traite le Soit comme si c’était un cas de base en lui « extrayant » son environnement local. La variable s’évalue dans Soit(v,w,u) en 2 temps. Donc la longueur de la liste est de taille [Environnement de base] + [nombre d’occurrence du constructeur Soit dans l’Environnement de base] donc au plus 2*[Environnement de base] donc le filtre se termine bien.