TP3: Types inductifs

MPSI Lycée Pierre de Fermat

Préliminaire : jeux de tests

Lorsque l'on écrit une fonction (en OCaml, en Python, ou autre), on doit toujours la tester pour vérifier qu'elle fonctionne bien. Plutôt que de faire les tests dans utop, on les mettra directement dans les fichiers sources `.ml`, sous la forme de **jeux de tests**. Pour cela, on peut utiliser la fonction `assert: bool -> unit`, qui fonctionne comme en Python : elle prend en entrée un booléen et arrête le programme s'il vaut `false`. Par exemple pour tester la fonction `len` calculant la longueur d'une liste, on pourra créer une fonction `test_len: unit -> unit` :

```
let test_len () =
  assert (len [] = 0);
  assert (len [2; 4; 5; 1; 1] = 5)
```

Si la fonction s'exécute sans erreur d'assertion, c'est que tous les tests ont réussi. Dans ce TP (et les suivants), écrivez systématiquement une petite batterie de tests pour chaque fonction implémentée. Idéalement, on écrira même les tests **avant** d'implémenter la fonction, afin de pouvoir directement tester et corriger rapidement. Prendre cette habitude vous permettra sur le long terme de gagner beaucoup de temps en TP (et donc aux épreuves !). On écrira aussi une fonction `test: unit -> unit` permettant de lancer tous les tests, de façon à pouvoir lancer les tests dans utop en tapant simplement `test ();;`.

1 La soif

On veut créer un type permettant de représenter les boissons (non-alcoolisées) avec :

- de l'eau ;
- du jus de fruit (il faut préciser quel fruit) ;
- du Breizh Cola, qui peut être normal ou light ;
- un cocktail de deux boissons (on devra préciser la proportion de la première boisson via un réel $p \in [0;1]$).

Q1. Créer un type `type boisson = ...` permettant de représenter les boissons. Créer quelques éléments de ce type et dessiner leurs arbres syntaxiques.

Q2. Créer une fonction qui calcule le prix au litre d'une boisson. On pose :

- L'eau est gratuite ;
- Tous les jus coûtent 3€ le litre, sauf le jus de ramboutan qui coûte 5.30€ le litre ;
- Le Breizh Cola coûte 1€ le litre.

Q3. Créer une fonction `shaker: boisson list -> boisson` prenant en entrée une liste non-vide de boissons $B_1 ... B_n$ et faisant un gigantesque cocktail, de la forme :

$$\textbf{cocktail}(\frac{1}{2}, B_1, \textbf{cocktail}(\frac{1}{2}, B_2, \textbf{cocktail}(...\textbf{cocktail}(\frac{1}{2}, B_{n-1}, B_n)...)$$

On voudrait pouvoir afficher la recette d'un cocktail[1], sous la forme :

```
Recette pour 1L:
50 mL Eau
400 mL Jus de raisin
300 mL Breizh Cola
250 mL Jus d'orange
```

En particulier, même si le cocktail a plusieurs instances d'un même ingrédient, la recette écrite devra les regrouper.

Dans la suite, on appelle **boisson de base** toute boisson n'étant pas un cocktail.

Q4. Écrire une fonction `string_of_boisson` calculant le nom d'une boisson de base.

Q5. Écrire une fonction `ingredients: boisson -> float -> (boisson*float)list` telle que pour b une boisson et $v \in R^+$ un volume en litres, `ingredients b v` renvoie une liste de couples (B, x) où B est une boisson de base, et x la quantité de cette boisson dans v litres de b. **On s'autorisera à avoir des doublons**, par exemple :

```
ingredients (Cocktail(Eau, Cocktail (Breizh true, Jus "pomme", 0.4), 0.5)) 1.;;
(* Vaut [(Eau, 0.5); (Breizh true, 0.2); (Jus "pomme", 0.3)] *)

ingredients (Cocktail(Eau, Eau, 0.5)) 0.4 ;;
(* Vaut [(Eau, 0.2); (Eau, 0.2)] *)
```

Q6. Énoncer le principe d'induction sur les boissons, puis montrer la propriété suivante par induction structurelle :

Pour toute boisson b, pour tout volume v, en notant $L =$ **ingredients(b, v)**, la somme des volumes de L vaut v.

Q7. Écrire une fonction de tri de liste (insertion, sélection, fusion).

En OCaml, les types somme possèdent un ordre par défaut. Pour comparer deux éléments d'un même type somme, OCaml commence par comparer les constructeurs utilisés, et s'ils sont égaux, alors c'est les paramètres qui sont comparés. Par exemple, lorsque l'on trie une liste de boissons, les boissons ayant les même constructeurs (les Eaux, les Jus, etc...) seront regroupés. Ainsi, si l'on trie la liste renvoyée par la fonction `ingredients`, on obtiendra une liste de couples (boisson, quantité) regroupés par boisson :

```
let l = [(Eau, 0.5); (Jus pomme, 0.3); (Eau, 0.2)];;
assert (select_sort l = [(Eau, 0.2); (Eau, 0.5); (Jus pomme, 0.3)]) ;;
```

Q8. Écrire une fonction `agreg_sum: ('a * float)list -> ('a * float)list` qui prend en entrée une liste de couples, et qui regroupe les couples selon la première composante, en sommant les deuxième composantes. On supposera que la liste d'entrée est triée. Par exemple :

```
agreg_sum [("bla", 0.1); ("bla", 0.3); ("truc", 0.4);("truc",0.2)] ;;
(* Vaut [("bla", 0.4); ("truc", 0.6)] *)
```

Indication : on pourra utiliser une fonction auxiliaire ayant deux paramètres supplémentaires donnant l'élément actuel et la somme actuelle pour cet élément.

Q9. En vous aidant des fonctions précédentes, écrire une fonction `recette: boisson -> unit` qui affiche la recette pour réaliser un litre d'une boisson, selon le format décrit plus haut.

1. Ne pas reproduire le cocktail donné en exemple chez vous

2 Expressions arithmétiques

OCaml se prête bien à l'écriture de petits compilateurs et interpréteurs, car les types sommes permettent assez facilement de représenter et de manipuler les arbres syntaxiques. On propose pour commencer le type suivant pour représenter des expressions simples :

```
type expr =
  | Const of float (* constante *)
  | Add  of expr * expr (* Add(e1, e2) correspond àe1 + e2 *)

(*** Exemples: ***)
(* représentation de 3.2 + 4 *)
let e1 = Add(Const 3.2, Const 4.)

(* représentation de (1 + 2) + (3 + (4 + 5)) *)
let e2 =
Add(
  Add(Const 1., Const 2.),
  Add(
    Const 3.,
    Add(Const 4., Const 5.)
  )
)
```

Notons que bien que l'on a appelé les constructeurs Const et Add, OCaml n'a aucune idée que ce sont des expressions arithmétiques. C'est la différence entre la **syntaxe** (la forme) et la **sémantique** (le sens). La définition du type est purement syntaxique, et la question suivante va permettre de définir la sémantique :

Q1. Écrivez une fonction `eval: expr -> float` qui évalue une expression. Par exemple :

```
assert (eval e1 = 7.2);;
assert (eval e2 = 15.);;
```

Q2. Ajoutez un constructeur `Mul` au type `expr`, servant à représenter le produit de deux expressions, et modifiez la fonction `eval` en conséquence.

Nous allons rajouter à nos expressions la possibilité d'avoir des variables, en rajoutant au type un constructeur `Var of string`. Pour évaluer une expression, il faut connaître la valeur des variables, c'est à dire fournir un **contexte**. Les listes de type `(string * float)list` serviront à représenter un ***contexte*** : si la liste L contient le couple (v, a), alors la variable nommée v sera associée à la valeur a :

```
type context = (string * float) list

(* contexte avec x -> 2.0, r -> -0.2 *)
let c1 = [("x", 2.0); ("r", -0.2)]
```

Q3. Écrire une fonction `get_var: string -> context -> int` telle que `get_var s l` cherche dans l un couple (s, n) et renvoie l'entier n correspondant. Cette fonction causera une erreur si la liste ne contient pas la variable recherchée.

La fonction précédente existe déjà dans OCaml : `List.assoc: 'a -> ('a * 'b)list -> 'b` prend en entrée un élément x et une liste L et renvoie le premier y tel que (x, y) apparaît dans x (et lève une erreur s'il n'y en a pas).

Q4. Modifier le type des expressions pour y rajouter le cas `| Var of string` représentant les variables. Modifier la fonction `eval` pour qu'elle prenne également un contexte en paramètre, et pour qu'elle traite le nouveau cas ajouté au type.

On souhaite maintenant augmenter nos expressions avec une construction if-then-else. Pour cela, nous allons créer un deuxième type, pour les expressions booléennes :

```
type boolexpr =
  | BConst of bool (* constantes true et false *)
  | Or of boolexpr * boolexpr (* OU booléen *)
  | Not of boolexpr (* NON booléen *)
  | Eq of expr * expr (* égalité *)
  | Leq of expr * expr (* inférieur ou égal *)
```

On rajoute également le constructeur suivant au type expr :

```
| IFTE of boolexpr * expr * expr (* if b then e1 else e2 *)
```

Comme les deux types dépendent l'un de l'autre, on doit les définir en utilisant le mot clé `and` :

```
type expr =
  ...
and boolexpr =
  ...
```

De la même manière, nous allons définir des fonctions sur ces deux types qui iront par paires et dépendront l'une de l'autre, il faudra alors aussi utiliser le mot clé **and**. Par exemple, les fonctions suivantes comptent le nombre d'occurrences d'une variable dans une expression / dans une expression booléenne :

```
let rec var_count (e: expr) (v: string) : int =
  match e with
  | Const _ -> 0
  | Var x -> if x = v then 1 else 0
  | Add (e1, e2) | Mul (e1, e2) -> var_count e1 v + var_count e2 v
  | IFTE (b, e1, e2) -> var_count_bool b v + var_count e1 v + var_count e2 v

and var_count_bool (b: boolexpr) (v: string) : int =
  match b with
  | BConst _ -> 0
  | Not bb -> var_count_vool bb v
  | Or (b1, b2) -> var_count_bool b1 v + var_count b2 v
  | Eq (e1, e2) | Leq (e1, e2) -> var_count e1 v + var_count e2 v
```

Q5. Modifier `eval` pour qu'elle soit définie en même temps qu'une fonction `eval_bool` équivalente.

Q6. Ajouter la possibilité de faire des `let in` (non récursifs) dans les expressions, en ajoutant un constructeur `Let of string * expr * expr` tel que `Let (v, e1, e2)` représente l'expression `let v = e1 in e2`.

Q7. (**Difficile**) Plutôt que d'avoir deux types distincts pour les expressions arithmétiques et booléennes, proposer un type unique `expr` fusionnant les deux. Expliquer les difficultés qui surviennent, et proposer d'éventuelles solutions. *Indication : il faudra peut être introduire de nouveaux types !*

Q8. (**Très difficile**) Ajoutez la possibilité de définir et d'utiliser des fonctions dans les expressions, puis des fonctions récursives.