Série d'exercices #5

IFT-2035

October 2, 2023

5.1 Mini-évaluateur en manque de fonctions

```
type Var = String

-- Expressions du code source en forme ASA.
data Exp = Enum Int           -- Une constante
         | Evar Var           -- Une variable
         | Elet Var Exp Exp   -- Une expr "let x = e1 in e2"
         | Ecall Exp Exp      -- Un appel de fonction

-- Valeurs renvoyées.
data Val = Vnum Int             -- Un nombre entier
         | Vprim (Val → Val)    -- Une primitive

elookup x ((x1,v1):env) =
   if x == x1 then v1 else elookup x env

eval env (Enum n) = Vnum n
eval env (Evar x) = elookup env x
eval env (Elet x e1 e2) =
   let v = eval env e1 in eval ((x,v):env) e2
eval env (Ecall fun actual) =
   case eval env fun of
     Vprim f -> f (eval actual)
```

Soit les déclarations ci-dessus utilisées pour un interpréteur:

1. Donner le type des deux fonctions. Vérifier que le code est typé correctement et corriger les éventuelles erreurs.

2. Étendre ce petit langage avec la possibilité de définir de nouvelles fonctions. I.e. Ajouter un constructeur d'expression `Elambda`, un constructeur de valeur correspondante `Vlambda`, ainsi que les cas pour gérer ces constructeurs dans `eval`.

5.2 Des fonctions pour p(l)aires

Définir en Haskell des fonctions pour représenter des *paires* en n'utilisant que des fonctions (pas de constructeur tels que `(:)`, `(,)`, ou de types algébriques). Plus précisément, définir:

`type` $P\ \alpha\ \beta = ...$		Le type des paires
`mkP`	:: $\alpha \to \beta \to P\ \alpha\ \beta$	Construction de paire
`leftP`	:: $P\ \alpha\ \beta \to \alpha$	Extraire l'élément de gauche
`rightP`	:: $P\ \alpha\ \beta \to \beta$	Extraire l'élément de droite

Ensuite, définir une fonction `unP` qui permet d'accéder à l'élément de gauche *et* celui de droite en une seule fois (plutôt qu'avec deux appels séparés à `leftP` et `rightP`). Bien sûr, toujours en n'utilisant que des fonctions.

Cela revient à créer une fonction qui d'une certaine manière "renvoie" deux valeurs, de même que le currying permet d'envoyer deux valeurs à une fonction.

5.3 Programmer sans variables

La programmation *point-free* ou *sans variable* consiste en un style de programmation fonctionnelle où l'on combine des fonctions pour en construire d'autres plutôt que d'utiliser la forme λ[1]. Les fonctions de base sont généralement appelées des *combinateurs*. En utilisant les fonctions prédéfinies telles que `(.)` (composition de fonctions, habituellement notée $\circ$), `(==)`, `(/)`, `not` ainsi que les fonctions ci-dessous:

```
flip f x y = f y x
map2 f (x:xs, y:ys) = f x y :  map2 f (xs, ys)
map2 f _ = []
```

mais sans utiliser λ (ni un λ implicite caché dans du sucre syntaxique, bien sûr), définir les fonctions suivantes:

```
pas_zero 2 ⟹ True      pas_zero 0 ⟹ False
moitie 13.2 ⟹ 6.6
somme [1,4,8] ⟹ 13
produit_scalaire ([1,2,3], [2,3,4]) ⟹ 20
```

Que vaut: `flip (.)  moitie sqrt 18` ?

[1] Le nom *point-free* vient du fait que les variables sont traditionnellement introduites dans le λ-calcul par des expressions qui utilisent une syntaxe où la variable est suivie d'un point, telles que $\lambda x.e$, $\forall x.e$, $\exists x.e$, ou $\mu x.e$