Projet Haskell – IR3
Le compte est bon

Stéphane Vialette

23 janvier 2017

Votre programme doit pouvoir être chargé par l'interpréteur ghci sans erreurs. La correction sera faite en appelant les fonctions demandées depuis ghci.

```
>: :load Countdown.hs
[1 of 1] Compiling Countdown         ( Countdown.hs, interpreted )
Ok, modules loaded: Countdown.
>:
```

Le travail est a faire en binôme. Vos nom doivent figurer en commentaire dans le code source.

1 Le problème *Le compte est bon*

Le problème *Le compte est bon* est défini de la façon suivante : étant donnés une liste d'entiers et un entier cible, décider s'il existe une expression arithmétique qui utilise au plus une fois chaque entier en entrée qui s'évalue à l'entier cible. Les entiers en entrée sont strictement positifs et il est de plus imposé que l'évaluation de toutes les sous-expressions arithmétiques durant les étapes intermédiaires fournissent également des entiers strictement positifs. Les opérations arithmétiques considérées sont : l'*addition*, la *soustraction*, la *multiplication* et la *division entière*.

Par exemple, pour la liste d'entiers `[1, 3, 7, 10, 25, 50]` et l'entier cible `765`, l'expression arithmétique $(1 + 50) \times (25 - 10)$ donne le résultat recherché. En fait, pour cet exemple, il existe 780 solutions. D'un autre coté, pour l'entier cible 831, il n'y a pas de solutions.

Le but est ici de proposer des solutions en haskell pour décider le problème *Le compte est bon*. Les types et fonctions définis en Section 2 doivent être nécessairement utilisés.

2 Quelques types et fonctions

Nous définissons un type `Op` pour les opérations arithmétiques et un prédicat `valid` qui décide si l'application d'un opérateur sur deux entiers non nuls est valide.

```haskell
-- arithmitic operations
data Op = Add | Sub | Mul | Div

instance Show Op where
  show Add = "+"
  show Sub = "-"
  show Mul = "*"
  show Div = "/"

valid :: Op -> Int -> Int -> Bool
valid Add _ _ = True          -- always true
valid Sub x y = x > y         -- do not allow for negative integers
```

```haskell
valid Mul _ _ = True          -- always valid
valid Div x y = mod x y == 0 -- integer division only
```

Nous définissons un type `Expr` pour les expressions arithmétiques et une fonction `values` qui retourne la liste des valeurs dans une expression arithmétique.

```haskell
data Expr = Val Int | App Op Expr Expr

instance Show Expr where
  show (Val x) = show x
  show (App op x y) = "(" ++ show x ++ show op ++ show y ++ ")"

values :: Expr -> [Int]
values (Val n) = [n]
values (App _ l r) = values l ++ values r
```

3 Brute force

En première approche, il s'agit de résoudre le problème *"Le compte est bon"* en générant et évaluant toute les expressions des entiers en entrée. Écrire la fonction `solution` qui retourne une expression arithmétique qui permet de résoudre le problème *"Le compte est bon"* (si une telle expresion existe, `Nothing` dans le cas contraire). Cette fonction génère toutes les expressions possibles pour tout sous-ensemble des entiers en entrée et sélectionne une qui est évaluée à l'entier cible.

```haskell
solution :: [Int] -> Int -> Maybe Expr
```

Pensez à définir des fonctions intermédiaires qui permettront de simplifier l'écriture de la fonction `solution` (par exemple, une fonction `eval` semble indispensable).

4 Élaguer l'arbre de recherche

Modifier la fonction `solution` (nous noterons `solutions'` la nouvelle fonction) pour tenir compte (liste non exhaustive ...) :

— de la commutativité de l'addition (*i.e.*, $x + y = y + x$),
— de la commutativité de la multiplication (*i.e.*, $x \times y = y \times x$),
— $1 \times x = x$, and
— $x/1 = x$.

5 Solutions approchées

Proposez une fonction `bestSolution` pour calculer une solution la plus proche de l'entier cible.

```haskell
bestSolution :: [Int] -> Int -> Expr
```