TD 9 : Non–déterminisme

1 Sommes de Rugby

L'inspiration de cet exercice est le sport de rugby et ses variations comme le football américain. Un aspect mathématiquement intéressant de ces sports est que certains scores sont impossibles, car une équipe qui marque de points peut marquer soit 3 soit 7 points [1]. Par conséquent, c'est impossible pour une équipe d'atteindre un score de 2, de 4, ou de 8 points, mais c'est possible d'atteindre un score de 16 point ($7+3+3+3$). C'est donc une question intéressante (facile à résoudre pour ces sports) quels sont les scores réalisables.

On aimerait généraliser ce problème : étant donné un ensemble d'entiers positifs, quels sont les sommes réalisables à partir de ces entiers ? On vous demande de programmer :

1. Une fonction `issum` qui prend comme paramètres un `Integer`, `n` et une liste de `Integer`, `xs` et retourne `True` si et seulement si c'est possible de construire la somme `n` en utilisant les entiers de la liste `xs` (autant de fois qu'on veut).
2. Une fonction `issum2` qui prend les mêmes paramètres que `issum` et retourne une `[[Integer]]`. Chaque élément de cette liste représente une manière d'atteindre `n` en utilisant les entiers de `xs` et la liste retourné doit être exhaustive (c'est-à-dire couvrir toutes les façons possibles d'atteindre `n`). Si `n` n'est pas atteignable on retourne la liste vide.
3. Une fonction `issum3` qui prend le mêmes paramètres que les fonctions précédentes et retourne un `Maybe [Integer]` : si `n` est atteignable on retourne `Just sol` où `sol` est une liste dont la somme est `n` et tous les éléments appartiennent à `xs` ; sinon, on retourne `Nothing`. (Vous pouvez utilisez les fonctions précédentes).
4. Une fonction `inksteps` qui retourne tous les scores possible qu'une équipe peut atteindre en marquant de points `k` fois. Cette fonction prend comme paramètre une liste `xs` qui contient les valeurs possibles si l'équipe marque de points une fois (pour le rugby `xs=[3,7]`), et `k`.

Pour les trois premières fonctions vous pouvez donner une implémentation récursive directe. Pour la fonction `inksteps` on vous propose de programmer une fonction secondaire avec la signature de type suivante : `step :: [Integer] -> [Integer] -> [Integer]`. Cette fonction prend comme premier paramètre la liste qui décrit les règles du sport (pour rugby `[3,7]`), et comme deuxième paramètre tous les scores qu'on a déjà calculé comme atteignables en marquant k fois. Elle retourne tous les scores qui sont atteignables en marquant $k+1$ fois. Il est conseillé d'utiliser le fait que les listes sont des applicatives et l'opérateur `<*>`. Il est également conseillé de faire attention à ne pas conserver de doublons (pourquoi ?).

Exemples d'exécution :

```
*Main> issum 16 [3,7]
True
*Main> issum 11 [3,7]
False
*Main> issum2 20 [5,6,9]
[[5,5,5,5],[5,6,9],[5,9,6],[6,5,9],[6,9,5],[9,5,6],[9,6,5]]
*Main> issum3 13 [5,6,9]
Nothing
*Main> inksteps [17,23] 6
[102,108,114,120,126,132,138]
```

1. Ceci n'est pas une description exacte des règles, mais ce n'est pas important pour cet exercice.

Solution :

```
issum :: Integer -> [Integer] -> Bool
issum 0 _ = True
issum n xs
 | n<0 = False
 | otherwise = or [ issum (n-i) xs | i<-xs ]


issum2 :: Integer -> [Integer] ->  [[Integer]]
issum2 0 _ = [ [] ]
issum2 n xs
 | n<0 = []
 | otherwise = [ i:sol | i<-xs, sol<-issum2 (n-i) xs ]

issum3 :: Integer -> [Integer] -> Maybe [Integer]
issum3 0 _ = Just []
issum3 n xs
 | issum n xs = Just $ head $ issum2 n xs
 | otherwise = Nothing


step :: [Integer] -> [Integer] -> [Integer]
step xs1 xs2 = (+) <$> xs1 <*> xs2

inksteps :: [Integer] -> Integer -> [Integer]
inksteps xs 1 = xs
inksteps xs n = qsort $ step xs (inksteps xs (n-1))

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (<x) xs)++[x]++qsort (filter (>x) xs)
```

2 Probabilités et Addition

On définit le type `type Proba=[(Int,Double)]` pour représenter une distribution de probabilité discrète (cf. examen partiel 2021). Une distribution est valide si la somme de toutes les probabilités est égale à 1 et on a pour chaque valeur possible `x` un seul élément `(x,p)` dans la liste.

Écrire une fonction `addition` qui prend comme arguments deux distributions de probabilité P_1, P_2 de type `Proba` et retourne la distribution de la variable aléatoire $Y = X_1 + X_2$, où X_1, X_2 sont deux variables indépendantes qui suivent les distributions P_1, P_2 respectivement.

Hint : Vous pouvez utiliser l'opération `<*>` pour générer une liste avec toutes les possibilités pour la somme $X_1 + X_2$. Puis, il faudra éliminer les doublons de cette liste pour produire une distribution valide.

Solution :

```
type Proba = [(Int,Double)]

cleanup :: Proba -> Proba
cleanup [] = []
cleanup ( (v,p):xs) = (v,p'):cleanup xs'
 where
  p' = p+ sum (map snd (filter (\(a,b) -> a==v) xs))
```

```
  xs' = filter (\(a,b) -> a/=v) xs

addition :: Proba -> Proba -> Proba
addition p1 p2 = cleanup $ (\(v1,p1) (v2,p2)->(v1+v2,p1*p2)) <$> p1 <*> p2
```

3 Rugby et Monads

On continue dans le même esprit que dans l'exercice précédent, mais cette fois on va utiliser le fait que les listes font partie de la classe `Monad` (et pas seulement la classe `Applicative`).

1. Écrire une nouvelle version de la fonction `step` avec type `step::[Integer]->Integer->[Integer]`. Le premier paramètre reste inchangé (pour rugby c'est `[3,7]`); le deuxième paramètre représente le score actuel. Le résultat retourné représente tous les scores suivants possibles. Donnez une version où vous utilisez l'opérateur `>>=` et une version où vous utilisez la notation `do`.
2. En utilisant cette nouvelle version de `step`, donnez une version alternative de la fonction `inksteps` de l'exercice précédent.

Solution :

Notez bien que la solution suivante est un bon exemple de la notion du non–déterminisme. La fonction `step` est écrite d'une façon qui pourrait vous amener à croire qu'elle retourne une `Integer` (on fait `return (x+xs2)` et `x+xs2` est bien un `Integer`). Or, la bonne manière d'interpréter cette fonction est comme une exécution non–déterministe : on va essayer toutes les valeurs `x<-xs1`; pour chaque telle valeur on calcule `x+xs2` et à la fin on retourne la concaténation de tous les résultats.

```
step :: [Integer] -> Integer -> [Integer]
--Les versions suivantes sont equivalentes
--step xs1 score = fmap (+score) xs1
--step xs1 score = xs1 >>= \x -> [score+x]
step xs1 score = do
                 x <- xs1
                 return (x+score)

inksteps :: [Integer] -> Integer -> [Integer]
inksteps xs 1 = xs
inksteps xs n = qsort $ inksteps xs (n-1) >>= step xs

qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort (filter (<x) xs)++[x]++qsort (filter (>x) xs)
```