Programmation fonctionnelle

Devoir surveillé final

27 mars 2017

Durée : 2h30
Documents (notes de cours, TD, TP et mini-haddock) autorisés

La clarté et la simplicité de vos réponses seront prises en compte dans l'évaluation. Dès que votre code n'est pas trivial, commentez-le ! En particulier, si vous définissez une fonction auxiliaire, spécifiez-la (que calcule-t-elle, quels arguments prend-elle, etc.).

Les questions marquées « ⋆ » sont plus difficiles, n'hésitez pas trop longtemps à passer à la suite.

Vous composerez votre DS sur deux copies séparées afin de paralléliser la correction.

1 Simplification d'expressions et analyse syntaxique

À composer sur la première copie

Nous souhaitons manipuler des expressions dans lesquelles peuvent apparaître des constantes, des variables et des opérateurs binaires ou unaires.

Structure de données

La structure de données que nous utiliserons sera la suivante :

```haskell
data Expression a = Valeur a
                  | Variable String
                  | Binaire (BinOp a) (Expression a) (Expression a)
                  | Unaire  (UnOp a)  (Expression a)
     deriving Show
```

Notez que le type de nos `Expression`s est paramétré par le type des valeurs. Nous pourrons ainsi manipuler tout aussi bien des expressions booléennes que des expressions entières ou des expressions de `Double`.

Pour les opérateurs, nous décidons de définir des types encapsulant à la fois la fonction de calcul correspondant à l'opérateur ainsi qu'une `String` pour l'afficher. Nous utiliserons donc le type suivant pour les opérateurs binaires :

```haskell
data BinOp a = B (a -> a -> a) String

instance Show (BinOp a) where
    show (B _ s) = s

opDeBinOp :: BinOp a -> a -> a -> a
opDeBinOp (B o _) = o
```

De même, pour les opérateurs unaires :

```haskell
data UnOp a = U (a -> a) String

instance Show (UnOp a) where
    show (U _ s) = s
```

Nous pouvons ainsi définir quelques opérateurs standard que vous pourrez utiliser dans la suite :

```haskell
plus  = B (+)         "plus"
moins = B (-)         "moins"
neg   = U (\x -> - x) "neg"

et  = B (&&) "et"
ou  = B (||) "ou"
non = U not  "non"
```

nous permettant de définir des expressions telles que :

```haskell
expr1 = Binaire plus (Binaire plus (Valeur 6) (Valeur 2))
                     (Binaire moins (Valeur 3) (Valeur 1))

expr2 = Binaire plus (Binaire plus (Valeur 6) (Valeur 2))
                     (Binaire moins (Variable "x") (Valeur 1))

expr3 = Binaire plus (Binaire plus (Valeur 6) (Variable "y"))
                     (Binaire moins (Variable "x") (Valeur 1))
```

correspondant aux expressions $(6+2)+(3-1)$, $(6+2)+(x-1)$ et $(6+y)+(x-1)$.

Q 1. En vous inspirant de `opDeBinOp`, donnez le type de la fonction `opDeUnOp` et définissez-la.

Q 2. Donnez les types inférés par GHC pour `plus`, `neg` et `non`.

Évaluation des expressions

Nous allons définir l'évaluation de ces expressions. Nous utiliserons le type `Maybe` pour traiter les cas où l'évaluation n'est pas possible (lorsqu'une expression contient une variable inconnue). Nous avons donc besoin de définir comment appliquer les opérateurs sur des valeurs de type `Maybe a`.

Q 3. Donnez la définition d'une fonction

```haskell
calculB :: BinOp a -> Maybe a -> Maybe a -> Maybe a
```

par exemple :

```
ghci> calculB plus (Just 12) Nothing
Nothing
ghci> calculB plus (Just 12) (Just 13)
Just 25
```

Vous pourrez aussi utiliser la fonction suivante sans la définir (elle est similaire à `calculB`, naturellement) :

```haskell
calculU :: UnOp a -> Maybe a -> Maybe a
```

Q 4. Donnez la définition d'une fonction

```haskell
eval :: Expression a -> Maybe a
```

évaluant son argument en retournant `Nothing` si l'expression contient une variable. Par exemple :

```
ghci> eval expr1
Just 10
ghci> eval expr2
Nothing
```

On ajoute maintenant un environnement pour associer des valeurs aux variables.

```haskell
type NomVariable = String
type Environnement a = [(NomVariable, a)]
```

Q 5. Donnez la définition d'une fonction

```
evalEnv :: Environnement a -> Expression a -> Maybe a
```

évaluant son argument expression en retournant `Nothing` si l'expression contient une variable qui n'est pas liée dans son argument environnement. On ne vérifiera pas qu'au plus une liaison est présente pour une variable donnée. Par exemple :

```
ghci> evalEnv [("x",2)] expr1
Just 10
ghci> evalEnv [("x",2)] expr2
Just 9
ghci> evalEnv [("x",2)] expr3
Nothing
```

Q 6. ⋆ Donnez la définition d'une fonction

```
evalSimp :: Environnement a -> Expression a -> Expression a
```

qui évalue tout ce qui peut s'évaluer au sens de la question précédente, c'est-à-dire :

— en remplaçant les variables définies dans l'environnement par leur valeur,
— en évaluant les opérateurs dès que c'est possible, c'est-à-dire quand les opérandes ont pu être complètement évalués.

Par exemple :

```
ghci> evalSimp [("x",2)] expr1
Valeur 10
ghci> evalSimp [("x",2)] expr2
Valeur 9
ghci> evalSimp [("x",2)] expr3
Binaire plus (Binaire plus (Valeur 6) (Variable "y")) (Valeur 1)
```

Malheureusement `evalSimp` ne permet pas de simplifier l'expression `Binaire ou (Valeur True) (Variable "x")` dans un environnement vide parce que `Variable "x"` ne peut pas être simplifiée.

Nous aurions besoin pour traiter ces cas-là d'une fonction spécifique de simplification pour chaque opérateur ; dans l'exemple de l'opérateur `ou`, cette fonction prendrait en argument les deux sous-expressions aussi simplifiées que possibles et retournerait l'expression la plus simplifiée possible.

Q 7. ⋆ Proposez une mise en place de cette simplification avancée :

— modifiez le type `BinOp` pour y ajouter une fonction de simplification,
— implémentez la fonction de simplification pour `ou`,
— modifiez la fonction `evalSimp` pour utiliser cette fonction de simplification par opérateur.

Analyse syntaxique

Nous allons maintenant développer un analyseur syntaxique pour des expressions booléennes. Pour éviter de gérer la question du parenthésage, nous utiliserons une notation polonaise, où les opérateurs sont préfixés.

La syntaxe sera la suivante :

— les *mots-clefs* `vrai` et `faux` désigneront les deux valeurs booléennes,
— les *mots-clefs* `et`, `ou` et `non` désigneront les opérateurs,
— les autres mots (suite de lettres) seront interprétés comme des variables,
— les différents mots seront séparés par des (au moins un) espaces.

Voilà le comportement que nous aurons pour l'analyseur `expressionP` à la fin :

```
ghci> runParser expressionP "vrai"
Just (Valeur True,"")
ghci> runParser expressionP "et vrai faux"
Just (Binaire et (Valeur True) (Valeur False),"")
```

```
ghci> runParser expressionP "x"
Just (Variable "x","")
ghci> runParser expressionP "ou faux y"
Just (Binaire ou (Valeur False) (Variable "y"),"")
ghci> runParser expressionP "ou et faux z vra"
Just (Binaire ou (Binaire et (Valeur False) (Variable "z")) (Variable "vra"),"")
```

Q 8. Définissez un parseur `espaces1P :: Parser ()` qui échoue si l'entrée ne commence pas par un espace et consomme tous les espaces initiaux.

```
ghci> runParser espaces1P  ""
Nothing
ghci> runParser espaces1P  " "
Just ((),"")
ghci> runParser espaces1P  " a"
Just ((),"a")
ghci> runParser espaces1P  "    abcd"
Just ((),"abcd")
```

Q 9. Définissez un parseur `motP :: Parseur String` qui consomme le premier mot (c'est-à-dire une suite de lettres, avec au moins une lettre) de l'entrée et échoue si l'entrée ne commence pas par un mot.

```
ghci> runParser motP ""
Nothing
ghci> runParser motP " abc"
Nothing
ghci> runParser motP "abc def"
Just ("abc"," def")
```

Q 10. Définissez un parseur `nonP :: Parser (UnOp Bool)` qui consomme le mot-clef `non` et retourne l'opérateur correspondant.

```
ghci> runParser nonP "no"
Nothing
ghci> runParser nonP "non"
Just (non,"")
ghci> runParser nonP "nonn"
Nothing
```

Le parseur pour les opérateurs binaires `ou` et `et` sera très similaire, tout comme celui pour les valeurs `vrai` et `faux`. Nous allons donc plutôt définir un parseur générique pour les mots-clefs de sorte à pouvoir définir les parseurs `binP` et `valP` (que vous utiliserez dans la suite) de la façon suivante :

```
binP :: Parser (BinOp Bool)
binP = motClefP [("et", et), ("ou", ou)]

valP :: Parser Bool
valP = motClefP [("vrai", True), ("faux", False)]
```

Q 11. Définissez un parseur `motClefP :: [(String, a)] -> Parser a` prenant en argument une liste de mots-clefs possibles et le résultat associé, réussit si l'un des mots-clefs est trouvé et retourne le résultat correspondant.

```
ghci> runParser (motClefP [("abc",1), ("def", 2)]) "abcdef"
Nothing
ghci> runParser (motClefP [("abc",1), ("def", 2)]) "abc def"
Just (1," def")
ghci> runParser (motClefP [("abc",1), ("def", 2)]) "def"
Just (2,"")
```

Q 12. Définissez le parseur `motPasClefP :: [String] -> Parser String` qui prend en argument la liste des mots-clefs et consomme le premier mot de l'entrée si ce n'est pas un mot-clef.

```
ghci> runParser (motPasClefP ["non"]) "no"
Just ("no","")
ghci> runParser (motPasClefP ["non"]) "non"
Nothing
ghci> runParser (motPasClefP ["non"]) "nonn"
Just ("nonn","")
```

Q 13. Définissez les parseurs suivants :

- `binExpP :: Parser (Expression Bool)` qui reconnaît une expression commençant par un opérateur binaire, donc de la forme : opérateur binaire, espaces (au moins un), première sous-expression, espaces, deuxième sous-expression,
- `unExpP :: Parser (Expression Bool)` qui reconnaît une expression commençant par un opérateur unaire,
- `expressionP :: Parser (Expression Bool)` qui reconnaît une expression complète.

Nous allons maintenant rajouter les environnements à la syntaxe. Nous utiliserons la syntaxe illustrée par les exemples suivants :

```
ghci> runParser envP ""
Just ([],"")
ghci> runParser envP "x=vrai"
Just ([("x",True)],"")
ghci> runParser envP "x=vrai,y=faux,var=vrai"
Just ([("x",True),("y",False),("var",True)],"")
```

Un environnement est une séquence d'association de valeurs à des variables où :

- les associations sont séparées par des virgules,
- il y a un nombre quelconque d'associations,
- une association est un nom de variable suivi du signe « = » et d'une valeur booléenne.

Q 14. Définissez un parseur `assocP :: Parser (NomVariable, Bool)` qui reconnaît une association d'une valeur à une variable.

Q 15. Définissez un parseur `envP :: Parser Environnement` pour les environnements.

Enfin nous utiliserons la syntaxe suivante pour donner à la fois un environnement et une expression : un environnement, suivi du signe « : » et d'une expression.

Q 16. Définissez un parseur `envExpP :: Parser (Environnement, Expression)`.

```
ghci> runParser envExpP "x=vrai:et x faux"
Just (([("x",True)],Binaire et (Variable "x") (Valeur False)),"")
```

Q 17. Enfin, définissez une fonction `analyseEtSimplifie :: String -> Maybe (Expression Bool)` qui analyse syntaxiquement une chaîne comme un environnement et une expression et calcule la version simplifiée au maximum de cette expression. Elle retournera `Nothing` si l'analyse syntaxique échoue.

2 Le problème des huits dames généralisé

À composer sur la seconde copie

Rappelons que nous apprécierons l'usage de fonctions standard qui évite de redéfinir des fonctions récursives. En particulier, vous pourrez utiliser la fonction `any :: (a -> Bool) -> [a] -> Bool` qui prend en paramètre une fonction testant une condition et une liste. `any` retourne `True` si au moins une valeur respecte la condition. Par exemple :

```
ghci> any (1 ==) [0,1,2,3,4,5]
True
ghci> any (> 5) [0,1,2,3,4,5]
False
```

Présentation du problème

Dans cet exercice, nous allons résoudre le problème des huit dames généralisé. Une dame peut se déplacer d'un nombre quelconque de cases soit (voir figure 1a) :

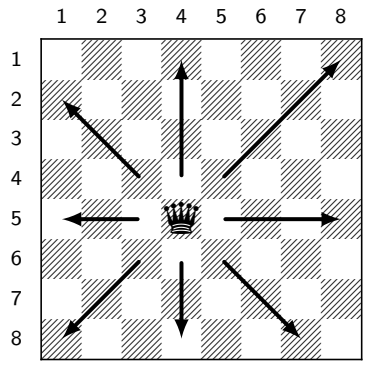
— horizontalement, en restant sur la même ligne,
— verticalement, en restant sur la même colonne,
— en diagonale.

Le but est de placer n dames sur un échiquier de $n \times n$ cases sans que celles-ci n'entrent en conflit entre elles, c'est-à-dire sans qu'aucune dame ne puisse atteindre la position d'une autre en un déplacement. Ainsi :

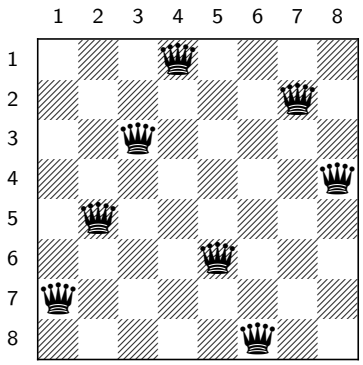
— il y a une et une seule dame sur chaque ligne,
— il y a une et une seule dame sur chaque colonne,
— il y a au plus une dame sur chaque diagonale.

Attention : par diagonale, nous n'entendons pas seulement les deux diagonales maximales (sur l'échiquier 8×8, celle de (1,1) à (8,8) et celle de (8,1) à (1,8)) mais aussi toutes leurs parallèles (par exemple de (3,1) à (1,3) ; de (1,5) à (4,8) ; etc.). Elles sont très nombreuses.

La figure 1b présente une des solutions possibles pour le problème à 8 dames.



(a) Déplacements possibles d'une dame



(b) Une solution du problème à 8 dames

FIG. 1: Règles

Représentation d'une solution

Puisqu'il doit y avoir exactement une dame sur chaque ligne, nous représenterons une solution par la liste des numéros de colonnes où elles sont placées. Par exemple, la solution donnée la figure 1b sera représentée par

la liste `[4,7,3,8,2,5,1,6]`. Par ailleurs, comme il ne peut y avoir deux dames dans la même colonne, cette liste sera une *permutation* des nombres `[1..n]`.

Cette représentation a l'avantage de toujours respecter les contraintes sur les lignes et sur les colonnes. Il ne reste donc que la contrainte sur les diagonales à vérifier.

Codage

Toutes les questions qui suivent sont indépendantes.

Dans un premier temps, nous allons définir une fonction `coord` pour passer d'une liste de numéros de colonnes à la liste des coordonnées (paires ligne, colonne) correspondante :

```
ghci> coord [3,1,2]
[(1,3),(2,1),(3,2)]
```

Q 18. Définissez la fonction `coord :: [Int] -> [(Int,Int)]` en utilisant une fonction auxiliaire récursive et du filtrage de motif.

Q 19. Définissez une fonction `coord' :: [Int] -> [(Int,Int)]` calculant le même résultat que `coord` mais utilisant la fonction `zip :: [a] -> [b] -> [(a,b)]` plutôt qu'une récursion.

Nous voulons maintenant détecter les conflits entre dames. Comme dit plus haut, il n'est nécessaire que de détecter si elles sont sur la même diagonale.

Q 20. Définissez une fonction `conflict :: (Int,Int) -> (Int,Int) -> Bool` qui prend deux coordonnées et retourne `True` si les deux coordonnées sont sur la même diagonale, `False` sinon.

```
ghci> conflict (1,4) (2,5)
True
ghci> conflict (1,4) (2,7)
False
ghci> conflict (1,4) (4,1)
True
```

Q 21. Définissez une fonction `conflicts :: (Int,Int) -> [(Int,Int)] -> Bool` qui prend les coordonnées d'une dame et une liste de coordonnées correspondant à celles des autres dames. Cette fonction retourne `True` si la dame est en conflit une autre dame de la liste, `False` sinon.

Q 22. Définissez une fonction `isValid :: [(Int,Int)] -> Bool` qui prend en argument les coordonnées de dames et retourne `True` si aucune des dames n'est en conflit avec une autre, `False` sinon.

```
ghci> isValid [(1,1),(2,3),(3,2)]
False
ghci> isValid [(1,1)]
True
```

Q 23. Définissez une fonction `solve :: Int -> [[Int]]` qui prend un nombre n et retourne la liste de toutes les solutions du problème des n dames.

Vous utiliserez pour cela la fonction `permutations :: [a] -> [[a]]` qui calcule l'ensemble des permutations possibles d'une liste.

```
ghci> permutations [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
ghci> solve 3
[]
ghci> solve 4
[[2,4,1,3],[3,1,4,2]]
```

Nous allons finir en définissant la fonction `permutations`.

Q 24. Définissez une fonction `insere :: Int -> [Int] -> [[Int]]`, qui prend un entier et une liste et qui retourne la liste de toutes les insertions possibles de l'entier en paramètre.

```
ghci> insere 4 [3,1,2]
[[4,3,1,2],[3,4,1,2],[3,1,4,2],[3,1,2,4]]
```

Trouver toutes les permutations des nombres de `[1..n]` revient à insérer n à toutes les positions possibles des permutations des nombres de `[1..n-1]` (ou, de façon équivalente, à insérer 1 dans `[2..n]`).

Q 25. Définissez la fonction `permutations :: [Int] -> [[Int]]`.

Q 26. ⋆ Définissez une fonction `permutations' :: [Int] -> [[Int]]` en utilisant uniquement une liste en compréhension et la fonction `delete :: a -> [a] -> [a]` qui supprime un élément d'une liste.