Programmation fonctionnelle

Devoir surveillé final

26 mars 2015

Durée : 2h
Documents (notes de cours, TD et TP) autorisés

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

1 Première copie

À composer sur la première copie

1.1 Correspondance de Curry-Howard

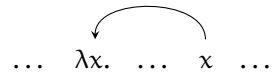
Q 1. En utilisant la correspondance de Curry-Howard, traduisez la phrase suivante en termes de logique : « `\x y -> y` est une fonction de type `a -> b -> b`. »

1.2 λ-calcul

Considérons les λ-termes suivants.

— $M_1 = \lambda x.x$,
— $M_2 = (\lambda x.x)y$,
— $M_3 = (\lambda x.(\lambda x.x)x)x$,
— $M_4 = (\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx))(\lambda x.x)$,
— $M_5 = (\lambda xyz.xy(yz))(\lambda xy.y)$.

Q 2. Pour chaque λ-terme, indiquez quelles sont les occurrences libres et les occurrences liées des variables. Pour les occurrences liées, vous indiquerez aussi où elle est liée par une flèche, par exemple :



Q 3. Pour chaque λ-terme, explicitez toutes les suites de réductions possibles en indiquant à chaque étape l'ensemble des rédex et la substitution à faire pour réduire l'expression. Vous préciserez en particulier leur forme normale, s'ils en ont une.

1.3 Structures de données

Nous voulons ici définir une structure de données pour des arbres binaires tels que :

— une feuille porte une valeur,
— un nœud ne porte pas de valeur, mais juste deux sous-arbres.

Le type des valeurs sera un paramètre de la structure de données.

Q 4. Déclarez le type `ArbreBinaire`.

Nous voulons extraire la valeur maximale d'un arbre binaire.

Q 5. Indiquez le type de la fonction `maxArbreBin` qui calcule la valeur maximale de l'arbre binaire passé en argument.
Définissez la fonction `maxArbreBin`.

Nous partons maintenant sur une définition alternative des arbres binaires, où :

— les feuilles ne portent plus de valeur,
— chaque nœud porte une valeur et deux sous-arbres.

Q 6. Déclarez le type `ArbreBinaire'`.

Q 7. Indiquez la difficulté que cette définition pose pour le calcul de la valeur maximale dans un arbre.
Proposez une solution et implémentez `maxArbreBin'`.

2 Seconde copie

À composer sur la seconde copie

2.1 Tri fusion

L'objectif de cet exercice est d'implémenter le tri fusion de listes.

Q 8. Définissez une fonction `decoupe :: [a] -> ([a],[a])` qui partage une liste en deux listes de longueurs presque égales en mettant un élément sur deux dans chaque liste. La première liste du résultat contiendra un élément de plus que la seconde si la liste de départ est de longueur impaire.

```
ghci> decoupe [1,2,3,4]
([1,3],[2,4])
ghci> decoupe [1,2,3,4,5]
([1,3,5],[2,4])
ghci> decoupe [1,2,3,4,5,6]
([1,3,5],[2,4,6])
```

Le tri *fusion* tire son nom de son opération fondamentale qui est la fusion de deux listes triées en une seule liste triée. Cette opération s'exprime naturellement de façon récursive : la fusion de deux listes (par exemple `[1,3,5]` et `[2,4,6]`) commence par le plus petit élément parmi les têtes des listes (ici `1`, donc), suivi de la fusion des restes (ici `[3,5]` et `[2,4,6]`).

Q 9. Définissez une fonction `fusionne :: Ord a => [a] -> [a] -> [a]` qui prend en argument deux listes triées et retourne leur fusion.

```
ghci> fusionne [1,4,7,10] [5,6,7,8]
[1,4,5,6,7,7,8,10]
```

Le tri fusion procède de la façon suivante : pour une liste d'au moins deux éléments (sans quoi la liste est forcément déjà triée !),

— la liste est découpée en deux sous-listes de longueurs presque égales,
— les deux sous-listes sont triées récursivement, puis fusionnées.

Q 10. Définissez une fonction `triFusion :: Ord a => [a] -> [a]`.

Nous allons maintenant tester que la fonction trie correctement. Rappelons la fonction `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]` qui prend en argument :

— une fonction `f`,
— une liste `[x1, x2, ...]`,
— une seconde liste `[y1, y2, ...]`,

et retourne la liste des `[f x1 y1, f x2 y2, ...]` (ayant la même longueur que la plus courte des deux listes en argument).

Q 11. Définissez une fonction `inferieurOuEgalAdjacents :: Ord a => [a] -> [Bool]` qui, étant donnée une liste `[x0,x1,x2,...]`, calcule la liste de booléens `[b01,b12,...]` telle que `b01` est `True` si et seulement si `x0` est inférieur ou égal à `x1`, `b12` est `True` si et seulement si `x1` est inférieur ou égal à `x2`, etc.

```
ghci> inferieurOuEgalAdjacents [0,1]
[True]
ghci> inferieurOuEgalAdjacents [1,0]
[False]
ghci> inferieurOuEgalAdjacents [1,0,0,1]
[False,True,True]
```

Vous proposerez une définition récursive et une définition utilisant `zipWith`.

Pour vérifier qu'une liste est triée en utilisant `inferieurOuEgalAdjacents`, nous avons besoin de replier la liste.

Q 12. Définissez une fonction `et :: [Bool] -> Bool` dont le résultat est `True` si et seulement si la liste ne contient aucun `False` (elle peut en particulier être vide).

Vous proposerez une version récursive et une version utilisant `foldr`.

Q 13. Définissez une fonction `triee :: Ord a => [a] -> Bool` qui calcule si la liste passée en argument est triée en combinant les fonctions précédentes.

Q 14. Définissez une propriété `prop_triFusion` que l'on pourra soumettre à QuickCheck et qui exprimera que le résultat de `triFusion` est une liste triée.

2.2 Analyse syntaxique

Dans cette partie, vous pouvez utiliser les analyseurs syntaxiques utilisés en TP, dont l'interface est rappelée en annexe.

Nous allons faire de l'analyse syntaxique d'un sous-ensemble du français. Les seules phrases autorisées seront celles qui se terminent par un point et qui sont de l'une des trois formes suivantes :

— un sujet suivi d'un verbe (exemple : « Je mange. »),
— un sujet suivi d'un verbe et d'un complément (exemple : « Je vois Alice. »),
— un sujet suivi d'un verbe et d'une subordonnée (exemple : « Je crois que je vole. »).

Vous utiliserez les types :

```haskell
type Sujet      = String
type Verbe      = String
type Complément = String

data Phrase = SV Sujet Verbe
            | SVC Sujet Verbe Complément
            | SVP Sujet Verbe Phrase
            deriving Show
```

Q 15. Définissez un analyseur syntaxique `mot :: Parser String` qui analyse le mot en début de phrase (s'il y en a un et s'il est différent de la conjonction « que »).

Par exemple :

```
ghci> parse mot ""
Nothing
ghci> parse mot "travaillent"
Just ("travaillent","")
ghci> parse mot "Ils   travaillent"
Just ("Ils","   travaillent")
```

```
ghci> parse mot "pomme    "
Just ("pomme","    ")
ghci> parse mot "   Bob"
Nothing
ghci> parse mot "que je mange"
Nothing
ghci> parse mot "   que je mange"
Nothing
```

Q 16. Définissez un analyseur syntaxique `espaces :: Parser ()` qui consomme tous les espaces au début de la chaîne à analyser, et retourne `()` pour ignorer ces espaces ou échoue si la phrase ne commence pas par un espace.

Par exemple :

```
ghci> parse espaces ""
Nothing
ghci> parse espaces " "
Just ((),"")
ghci> parse espaces "     "
Just ((),"")
ghci> parse espaces "  a "
Just ((),"a ")
```

Q 17. Compléter les définitions d'analyseurs syntaxiques :

```
fin :: Sujet -> Verbe -> Parser Phrase
fin s v = ...

complement :: Sujet -> Verbe -> Parser Phrase
complement s v = ...

subordonnee :: Sujet -> Verbe -> Parser Phrase
subordonnee s v = ...

phrase :: Parser Phrase
phrase = ...
```

`fin s v` analyse une fin de phrase (c'est-à-dire un point) et lui rajoute le sujet `s` et le verbe `v`.

`complement s v` analyse un complément (c'est-à-dire : au moins un espace, suivi d'un mot puis d'un point qui finit la phrase) et lui rajoute le sujet `s` et le verbe `v`.

`subordonnee s v` analyse une subordonnée (c'est-à-dire : au moins un espace suivi de « que » puis d'au moins un espace et d'une phrase) et lui rajoute le sujet `s` et le verbe `v`.

`phrase` analyse une phrase dans l'une des trois formes autorisées.

Par exemple :

```
ghci> parse (fin "Je" "mange") ".   "
Just (SV "Je" "mange","   ")
ghci> parse (fin "Je" "mange") "  ."
Nothing
ghci> parse (fin "Je" "vois") "Alice."
Nothing

ghci> parse (complement "Je" "vois") "   Alice.  Je pense."
Just (SVC "Je" "vois" "Alice","  Je pense.")
ghci> parse (complement "Je" "vois") "   Alice"
```

```
Nothing
ghci> parse (complement "Je" "vois") "Alice."
Nothing

ghci> parse (subordonnee "Je" "crois") "   que je vole.  "
Just (SVP "Je" "crois" (SV "je" "vole"),"  ")
ghci> parse (subordonnee "Je" "crois") "   que je vole"
Nothing
ghci> parse (subordonnee "Je" "crois") "que je vole."
Nothing
ghci> parse (subordonnee "Je" "vois") "   Alice.  Je pense."
Nothing

ghci> parse phrase "Je pense que tu sais que je viendrai demain. Je crois."
Just (SVP "Je" "pense" (SVP "tu" "sais" (SVC "je" "viendrai" "demain")),
 " Je crois.")
```

Annexe : Rappels sur quelques analyseurs syntaxiques élémentaires

Le type `Parser` est une instance de la classe `Monad`, de sorte qu'il est possible d'utiliser la notation `do` en plus de l'utilisation directe de `return` et `(>>=)`.

- `return :: a -> Parser a` est la fonction qui, lorsqu'elle est appliquée à une valeur, renvoie l'analyseur syntaxique trivial qui retourne toujours cette valeur en ne parsant rien.
- `(>>=) :: Parser a -> (a -> Parser b) -> Parser b` met en séquence deux analyseurs.
- `caractere :: Parser Char` parse le premier caractère de la chaîne.
- `echoue :: Parser a` est l'analyseur qui échoue toujours.
- `(|||) :: Parser a -> Parser a -> Parser a` est le combinateur qui retourne le résultat du premier analyseur si celui-ci réussit, sinon celui du second.
- `carCond :: (Char -> Bool) -> Parser Char` parse un caractère qui vérifie une condition donnée.
- `car :: Char -> Parser Char` parse un unique caractère.
- `chaine :: String -> Parser String` parse une chaîne constante.
- `zeroOuPlus :: Parser a -> Parser [a]` combine des parseurs pour des séquences de zéro ou plusieurs éléments.
- `unOuPlus :: Parser a -> Parser [a]` combine des parseurs pour des séquences d'au moins un élément.