Principes de Programmation
TP1

21 janvier 2019

Il y a un sondage disponible sur la page du cours. Vous pouvez le faire maintenant ou plus tard (le contenu peut être sensible).

Exercice 1 (Premier contact).

1. Lancez l'interpréteur *ghci* (tapez `ghci` dans un terminal...).
2. Tapez `3 == 4` et entrez, qu'obtient-on ?
3. Tapez `1:2:3:[]`, qu'obtient-on ?
4. Tapez `'a':'b':'c':[]`, qu'obtient-on ? [1]
5. Demandez le type de `False` en tapant la commande `:t False`.
6. Tapez `:t not`, qu'obtient-on ?
7. Tapez `:t (&&)`, qu'obtient-on ?
8. Tapez `:t (||)`, qu'obtient-on ?
9. Tapez `:t (&& True)`, qu'obtient-on ? Pourquoi ?
10. Tapez `:t map`, qu'en déduisez vous sur ce que fait map ?
11. Tapez `map (&& True) [True,False,True,True,True]`, qu'obtient-on ?

Exercice 2 (Prédicats logiques).
Un prédicat logique clos est décrit par : [2]

```
data PredClos =   VRAI | FAUX
                | OU PredClos PredClos
                | ET PredClos PredClos
                | NON PredClos
                | IMPL PredClos PredClos
               deriving (Eq,Show)
```

Cela veut dire qu'un prédicat logique clos est l'un des 6 :
— une constante `VRAI`,

1. En Haskell, une string est juste une liste de caractère. Les problèmes d'optimisation sont laissés au compilateur.
2. La ligne "`deriving (Eq,Show)`" sera expliqué plus tard dans le cours.

— une constante `FAUX`,
— un couple `OU p1 p2` où `p1` et `p2` sont des prédicats logiques clos,
— un couple `ET p1 p2` où `p1` et `p2` sont des prédicats logiques clos,
— un singleton `NON p` où `p` est un prédicat logique clos.
— ou un couple `IMPL p1 p2` où `p1` et `p2` sont des prédicats logiques clos,

1. Écrivez cette définition de *type algébrique* dans un fichier `TP1.hs` et chargez dans *ghci* à l'aide de la commande :
   ```
   Prelude> :load TP1.hs
   ```
2. Dans *ghci*, testez
   ```
   Prelude> let exemple = OU (IMPL VRAI FAUX) (NON FAUX)
   Prelude> exemple
   ```
   Un prédicat logique clos est une structure de donnée similaire à un arbre avec des opérations logique sur les noeuds et feuilles. C'est la version parsée mais pas évaluée d'une formule écrite par un utilisateur par exemple.
3. Écrivez la fonction suivante :
   ```
   impl :: PredClos -> PredClos -> PredClos
   impl p1 p2 = OU (NON p1) p2
   ```
   Que fait-elle ? Testez là.
4. Écrivez une fonction `equiv :: PredClos -> PredClos -> PredClos` qui implémente une équivalence logique.
5. Écrivez la fonction suivante :
   ```
   suprIMPL :: PredClos -> PredClos
   suprIMPL VRAI        = VRAI
   suprIMPL FAUX        = FAUX
   suprIMPL (OU   p1 p2) = OU   (suprIMPL p1) (suprIMPL p2)
   suprIMPL (ET   p1 p2) = ET   (suprIMPL p1) (suprIMPL p2)
   suprIMPL (IMPL p1 p2) = impl (suprIMPL p1) (suprIMPL p2)
   suprIMPL (NON  p    ) = NON  (suprIMPL p)
   ```
   Testez là. Que fait-elle ? Comment ? On appel ça du *pattern-matching*.
6. Dans *ghci*, tapez `:type not`, puis `:type (||)` et `:type (&&)`. Ce sont les types de la négation booléenne et des opérateurs de conjonctions et disjonctions.
7. Implémentez un évaluateur `eval :: PredClos -> Bool` évaluant la valeur de vérité d'un prédicat clos. Par exemple `eval exemple` doit rendre `True`.
8. (Difficile) [3] Un prédicat est la même chose qu'un prédicat clos sauf qu'il peut contenir des variables `Var Int` ; écrire le datatype *Pred*.

3. La difficulté vient ici de comprendre la syntaxe, n'hésitez pas à demandé au chargé de TP pour les questions difficiles.

9. (Difficile) Un environnement est une fonction `envi :: Int -> Bool` qui associe à chaque variable $Var\ n$ un booléen $envi(n)$.
   Écrire un évaluateur `evalPred :: Pred -> (Int->Bool) -> Bool` qui évalue un prédicat dans un environnement donné en paramètre.
   exemple : `evalPred (Ou (Var 1) (Var 2)) (==1)` rend le booléen `True` car dans l'environnement `(==1)`, le prédicat `Var 1` est vrai ; par contre `evalPred (OU (VAR 1) (VAR 2)) (>2)` rend `False`.
10. (Difficile) Écrire une fonction `variables :: Pred -> [Int]` donnant tous les indices des variables de son argument.
    exemple : `variables (Ou (Var 1) (Et (Var 42) (Var 12)))` rend la liste `[1,42,12]`.

1 À faire chez soi :

Exercice 3 (Tutoriel Officiel).
Allez sur le site officiel de Haskell, tapez `help` dans le prompteur et faites le tutoriel. L'anglais n'y est pas trop difficile, avec un traducteur automatique vous devriez comprendre la trame. Si un point d'anglais ou un résultat vous bloque vraiment consultez le professeur.

Un détail qui peut être déroutant sont les types affichés. Ce tuto ignore généralement les types qui sont d'un niveau un peu élevé mais ils perturbent, en particulier les notations du type `Num a => a` ou `Num a => [a]`. On les expliquera plus tard, pour l'instant considérez que cela veut dire les types `Int` ou `[Int]`, c'est à dire que "`Num a =>...`" se comprend comme "`a=Int` dans...". De même `Fractional a` veut dire "a est un flottant" et les conditions `Ord a` et `Enum a` peuvent être ignorés.

Installer Haskell Installez *Haskell* (déjà fait sur les postes de l'université) comme décrit dans la Section 1 des consignes d'installation.

Un IDE pour Haskell Nous vous conseillons d'utiliser l'IDE *Atom*. Il est normalement déjà installé sur les postes de l'université, et si vous devez l'installer chez vous vous pouvez suivre la Section 2 des consignes d'installation. L'intérêt d'*Atom* est de pouvoir se paramétrer pour *Haskell*, vous pouvez pour cela suivre la Section 3 des consignes d'installation. Attention selon la version de *Atom* installée, certains paramétrages ne pourront pas fonctionner et renverront des message d'erreur ; revenez en arrière dans ces cas (ou installez une version plus récente). Si la version est trop ancienne (et que l'on n'a pas les options d'*Atom*), il se peut qu'il n'y ai pas beaucoup d'intérêts à utiliser *Atom* plutôt qu'un éditeur plus classique (*emacs*, *Vim*, etc...) pourvu que l'on ai la coloration syntaxique.

Exercice 4 (Exceptions).
Nous verrons plus tard un autre mécanisme d'exception. Mais pour l'instant, vous pouvez vous contenter du mécanisme de base :

1. Lancez l'interpréteur *ghci* (tapez `ghci` dans un terminal...).
2. tapez `:type 5/0`, qu'obtient-on ?
3. tapez `5/0`, qu'obtient-on ?
4. essayez avec `[0..5] !! 6`,
5. avec `undefined`,
6. ou avec `error "Qu'attendiez-vous?"`

Exercice 5 (Suite des prédicats logiques (bonus+)).
Une conjonction de disjonction est une liste (vu comme un grand ET) de clauses, elles-mêmes étant des listes (vues comme des grands OU) d'atomes, c'est à dire de variables ou de négations de variables.

Le but de cet exercice est d'écrire une fonction mettant un prédicat logique sous sa forme cannonique.

1. Un prédicat aux négations atomiques est un prédicat où on ne peut prendre la négation que des variables. En terme de type algébrique, il n'y a plus de constructeur `NON`, mais il y a un nouveau constructeur `NONVAR` pour les variables négativées. Décrire le type algébrique `NAPred`.
2. Écrire l'évaluation d'un `NAPred` (avec un environement).
3. Écrire une fonction de décente des négation `descente :: Pred -> NAPred` qui transforme le prédicat en utilisant les rêgles logiques :

   $$\texttt{non}\,(p\,\texttt{et}\,q) \;=\; (\texttt{non}\,p)\,\texttt{ou}\,(\texttt{non}\,q) \qquad \texttt{non}\,(p\,\texttt{ou}\,q) \;=\; (\texttt{non}\,p)\,\texttt{et}\,(\texttt{non}\,q) \qquad \texttt{non}\,(\texttt{non}\,p) \;=\; p$$

4. Décrire les types de données `Dij` pour les disjonctions et `ConjDij` pour les conjonctions de disjonctions.
5. Écrire un évaluateur pour les conjonctions de disjonctions.
6. Écrire une fonction `napredTotoConjdij :: NAPred -> ConjDij` envoyant un prédicat aux négations atomiques sur sa représentation canonique en utilisant la rêgle logique :

   $$(p\,\texttt{et}\,q)\,\texttt{ou}\,r \;=\; (p\,\texttt{ou}\,r)\,\texttt{et}\,(q\,\texttt{ou}\,r)$$

7. Écrire une fonction `totoConjdij :: Pred -> ConjDij` envoyant un prédicat sur sa représentation canoniqu.