Programmation Fonctionnelle
Raisonner autour des programmes

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

7 octobre 2013

Plan

Le module `QuickCheck`

Raisonnement algébrique sur les programmes

`reverse`

`foldr` et `map`

Plan

Préambule

Parmi les objectifs de la programmation fonctionnelle, mentionnés au début du cours, celui de pouvoir créer des programmes

- surs,
- certifiés,
- efficaces.

Tests

Le module `QuickCheck` permet de tester la validité d'un prédicat sur un nombre d'instances engendrées aléatoirement.

```
import QuickTest

quickCheck ::
    Arbitrary a => (a -> Bool) -> IO ()
quickCheckWith ::
    Arbitrary a => Args -> (a -> Bool) -> IO ()
```

où `args` est de la forme

```
Args {
  maxSuccess :: Int,
  maxSize :: Int,
  ...
}
```

Tester le module `Complexe`

Dans le module `Complexe`, le type `Complexe` est déclare être une instance de la classe `Num`.

La classe `Num` possède les méthodes

```
abs :: a -> a,
signum :: a -> a
```

et on demande que la loi

```
abs x * signum x == x
```

soit toujours vérifiée.

Importation des modules nécessaires :

```
import Complexe
import Test.QuickCheck
import Test.QuickCheck.Arbitrary
```

On définit deux prédicats pour tester cette loi :

```
-- test naif
test_abssignum :: Complexe -> Bool
test_abssignum x =
    abs x * signum x == x

-- test modulo un petit erreur
test_abssignum_error ::
    Float -> Complexe -> Bool
test_abssignum_error err x =
    modulus ((abs x * signum x) - x) < err
```

La classe `Arbitrary`

On instancie `Complexe` à la classe `Arbitrary` pour pouvoir engendrer des complexes de façon aléatoire :

```
instance Arbitrary Complexe where
    arbitrary = do
      x <- arbitrary
      y <- arbitrary
      return (construct x y)
```

Remarque :

```
arbitrary :: Arbitrary a => Gen a
```

Voici la suite de tests

```
test1 = quickCheck test_abssignum
err = 0.001
test2 = quickCheck (test_abssignum_error err)
```

et les résultats :

```
*Main> test1
*** Failed! Falsifiable (after 2 tests):
-1.0942812 + i-2.498806
*Main> test2
*** Failed! Falsifiable (after 34 tests):
4052.5146 + i21.902075
*Main> test2
+++ OK, passed 100 tests.
*Main> test2
+++ OK, passed 100 tests.
*Main> test2
*** Failed! Falsifiable (after 27 tests):
-3031.4456 + i13.3034115
```

Les tests ne sont pas conclusifs :

- si le prédicat est falsifié,
  nous souhaitons connaître la largeur de l'erreur,
- nous souhaitons aussi engendrer un bien plus grand nombre de tests.

```
data Couple = Couple (Complexe,Float)
              deriving Show
instance Arbitrary Couple where
    arbitrary = do
      x <- arbitrary
      let y = modulus (abs x * signum x - x)
      return (Couple (x,y))

args = stdArgs { maxSuccess = 5000,
                                  maxSize = 10000}
test4 = quickCheckWith args (test_paire err)
        where
          test_paire err (Couple (x,y)) =
              y < err
```

Résultats :

```
*Main> test4
*** Failed! Falsifiable (after 80 tests):
Couple (-7580.3286 + i-2.288621,2.2874763)
*Main> test4
*** Failed! Falsifiable (after 40 tests):
Couple (18192.688 + i-102.07043,0.18349457)
*Main> test4
*** Failed! Falsifiable (after 45 tests):
Couple (6193.915 + i-61.179153,2.4608612e-2)
*Main> test4
*** Failed! Falsifiable (after 22 tests):
Couple (50408.86 + i70.76846,0.99224854)
*Main>
```

Conclusion : tout à refaire :-(

Plan

Le module `QuickCheck`

Raisonnement algébrique sur les programmes

`reverse`

`foldr` et `map`

Les tests sont ils suffisants ?

Le module `Complexe`, qui semblait être correcte selon une premiere suite de tests, a été detruit par une autre suite.

Pour un code critique – à lire :

- contrôleur d'un avion, ou du space-shuttle,
- système réservations SNCF,
- ...

– on peut avoir besoin de la certitude qu'un programme se comporte comme specifié.

Cet objectif est assez difficile à obtenir,
mais on peut se mettre en route.

Exemple : correction de `Pile`

Considérez la classe `Stack`, et son instance `Pile` :

```
class Stack m where
    (
     push :: a -> m a -> m a
     pop :: m a -> m a
    -- loi a satisfaire :
    -- pop (push x p) == p
    -- pour tout x et p
    )

newtype Pile a = Pile [a]
instance Stack Pile where
    push x (Pile xs) = Pile (x:xs)
    pop (Pile (x:xs)) = Pile x
```

Preuve

La loi est satisfaite :

```
pop (push x p)
  == pop (push x (Pile xs))
                (car p == Pile xs, pour quelque xs)

  == pop (Pile (x::xs))
                (par def. de push)

  == Pile xs
                (par def. de pop)

  == p
                (car p == Pile xs)
```

Vérification des programmes en Haskell

Les fonctions de Haskell étant définies par équations, on démontre des propriétés des programmes le plus prouvés en utilisant le raisonnement algébrique (qui procède par équations).

D'autres éléments habituels dans ce type de raisonnement (surtout avec les types récursifs) :

- utilisation de l'induction,
- raisonnement par cas, selon les constructeurs.

Remarques :

- Ce type de raisonnements sont valables pour le code *pure* (qui ne contient pas d'effets de bord),

- dans l'attente que Dassault, SNCF, ou l'armée nous recrutent, on peut souligner le fait suivant :

> *le raisonnement algébrique vient en aide pour améliorer la performance des programmes.*

Plan

Le module `QuickCheck`

Raisonnement algébrique sur les programmes

`reverse`

`foldr` et `map`

reverse

- La définition suivante de la fonction `reverse` :

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs++[x]
```

  n'est pa optimale, car elle demande temps quadratique dans la longueur de liste en paramètre.

- En fait, la fonction `(++)` est linéaire dans la longueur de son premier paramètre.

reverse_p

L'implantation suivante

```
reverse_p :: [a] -> [a]
reverse_p xs = reverse_acc xs []

reverse_acc :: [a] -> [a] -> [a]
reverse_acc [] ys = ys
reverse_acc (x:xs) ys = reverse_acc xs (x:ys)
```

est plus performante, car elle prends temps linéaire.

Mais le problème s'impose : est il vrai que

```
reverse xs == reverse_p xs
```

pour toute liste `xs` ?

reverse == reverse_p

Nous allons prouver ce fait, en utilisant une lois plus générale :

reverse_acc xs ys == reverse xs ++ ys (*)

pour toute liste `xs` (et toute liste) `ys`),

de façon que :

```
reverse_p xs
  == reverse_acc xs []
       (par def. de reverse_p)

  == reverse xs ++ []
       (par (*))

  == reverse xs
       (parce que ys++[] == ys, voir TD)
```

- Les listes possèdent deux constructeurs, `[]` et `:` .
- La preuve sera structurée par induction, selon ces deux cas.

Cas où le premier paramètre est la liste vide :

```
reverse_acc [] ys
  == ys
       (par def. de reverse_acc)

  == [] ++ ys
       (par def. de ++, a l'envers).
```

Cas ou le premier paramètre est une liste non vide, de la forme `x:xs`.
On pourra utiliser l'hypothèse d'induction :

```
reverse_acc xs ys == reverse xs ++ ys (HI)

reverseacc (x:xs) ys
  == reverse_acc xs (x:ys)
       (par def. de reverse_acc)

  == reverse xs ++ (x:ys)
       (par HI)

  == reverse xs ++ ([] ++ (x:ys))
       (par def. de (++), a l'envers)

  == reverse xs ++ ([x] ++ ys)
       (par def. de (++), a l'envers, encore)

  == (reverse xs ++ [x]) ++ ys
       ((++) est associatif, voir TD)

  == reverse (x:xs) ++ ys
       (par def. de reverse, a l'envers)
```

Plan

Le module `QuickCheck`

Raisonnement algébrique sur les programmes

`reverse`

`foldr` et `map`

foldr et *map*

Rappelons d'abord les définitions de ces fonctions :

```
foldr :: (a -> b -> b) ->  b ->  [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x:map f xs
```

On démontre que les deux espressions

```
foldr f v (map g xs)
foldr (\x y -> f (g x) y)) v xs
```

sont équivalentes.

Remarques :

- La deuxième expression est préférable,
  parce qu'elle ne demande pas d'allouer de la mémoire pour une liste intermédiaire (et de faire travailler le GC comme un fous).
- De même, on peut montrer que les deux expressions

  ```
  foldl f v (map g ys)
  foldl (\x y -> f x (g y)) v ys
  ```

  sont équivalentes.

```
foldr f v (map g [])
  ==  foldr f v []
  == v
  == foldr (\x y -> f (g x) y)) v []
```

```
foldr f v (map g (x:xs))

  ==  foldr f v (g x:map g xs)

  == f (g x) (foldr f v (map g xs))

  == f (g x) (foldr (\x y -> f (g x) y)) v xs)
     (par HI)

  == foldr (\x y -> f (g x) y) v (x:xs)
```

Application

Par exemple, en utilisant cette règle on peut simplifier le code suivant :

```
etoiles :: Int -> String
etoiles x = replicate x '*'

fun1 :: Int -> Int
fun1 n = foldr f 0 (map etoiles [1..n])
  where
    f z y = length z + y
```

en remplaçant `fun1` par

```
fun2 :: Int -> Int
fun2 n = foldr h 0 [1..n]
  where
    h x y = length (etoiles x) + y
```

et ensuite `fun2` par

```
fun3 :: Int -> Int
fun3 n = foldr (+) 0 [1..n]
```