Ficha7-ATS

Luís Braga - https://github.com/BragaMann

December 2019

Contents

1 Ficha7

```haskell
module Ficha7 where

import Test.QuickCheck
import Data.List
import System.Random

-- stack ghci --package QuickCheck -- .\Ficha7.hs

-- Exercicio 1. Tendo em conta a seguinte defini  c   ao  de mulL:
mulL :: Num a => [a] -> a
mulL [] = 0
mulL (h : t) = h * mulL t

-- defina as seguintes propriedades para a funcao:
-- a) a listagem inversa (reverse) da lista devolve o mesmo resultado;

prop_mulLa :: (Num a, Eq a) => [a] -> Bool
prop_mulLa l = mulL (reverse l) == mulL l

-- b) o resultado para uma lista com um elemento   e  igual a esse elemento;

prop_mulLb :: Int -> Bool
prop_mulLb a = mulL [a] == a

-- c) o resultado   e  igual ao resultado da fun  c   ao  product do Haskell.

prop_mulLc :: [Int] -> Bool
prop_mulLc l = mulL l == product l

-- Exercicio 2. Considere a seguinte definicao da funcao find

find :: (a -> Bool) -> [a] -> Maybe a
find f [] = Nothing
find f (x:xs) = case Ficha7.find f xs of
    Just k -> Just k
    Nothing -> if f x then Just x else Nothing
-- o erro e devolver o elemento mais a direita

-- que contem um erro na sua definicao.

-- a) Defina propriedades que a funcao deve ter e teste-as.

prop_find1 :: (Eq a) => (a -> Bool) -> [a] -> Bool
prop_find1 f l = Ficha7.find f l == Data.List.find f l


prop_find2 l = all (>0) l ==> Ficha7.find (==0) l == Nothing

prop_find3 l = length (filter odd l) > 0 ==> Ficha7.find (odd) l == Just (head (filter odd l))


-- b) Corrija o codigo e teste novamente as propriedades.

```

```haskell
-- The find function takes a predicate and a structure
-- and returns the leftmost element of the structure matching
-- the predicate, or Nothing if there is no such element.

find' :: (a -> Bool) -> [a] -> Maybe a
find' f [] = Nothing
find' f (x:xs) = if f x then Just x else find' f xs

prop_find1' :: (Eq a) => (a -> Bool) -> [a] -> Bool
prop_find1' f l = find' f l == Data.List.find f l

{-
Exercicio 3. Por vezes, comete-se o erro de se delegar o trabalho de geracao
a propriedade a ser testada em vez do gerador em si. Ha ferramentas que
permitem o uso condicional ou limitado de geradores, no entanto estas nao
devem ser abusadas. Pode-se limitar os valores testados atraves de uma pre-
condicao:
-}

--prop_positive :: Int -> Property
--prop_positive x = x > 0 ==> (x * x * x) > 0

-- Ou escolher qual o gerador a utilizar na propriedade:

--prop_positive' :: Property
--prop_positive' = forAll genPosInt $ \k -> (k * k * k) > 0

{-
Considere as seguintes funcoes para verificar se uma lista esta ordenada e para
inserir um elemento numa lista, respetivamente:
-}

sorted :: Ord a => [a] -> Bool
sorted l = and (zipWith (<=) l (tail l))

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
            | x<y = x:y:ys
            | otherwise = y:Ficha7.insert x ys

{-
1. Defina uma propriedade prop_ins_ord :: Int -> [Int] -> Property
que, dado um inteiro e uma lista de inteiros, verifica que, caso a lista de
inteiros esteja ordenada, ent  ao  essa lista continua ordenada ap  os  a inser  c  ao
do novo inteiro. Explique os resultados obtidos.
-}

-- S  se usa property por ser uma propriedade condicional
prop_ins_ord :: Int -> [Int] -> Property
prop_ins_ord x l = sorted l ==> sorted (Ficha7.insert x l)

{-
2. Acrescente a propriedade definida na al nea anterior o combinador collect,
que ira agrupar os resultados obtidos de acordo com um criterio rece-
bido como argumento. Sugestao: Agrupe os resultados de acordo com o
tamanho da lista recebida como argumento da propriedade.
-}
prop_ins_ord' :: Int -> [Int] -> Property
prop_ins_ord' x l = collect (length l) $ sorted l ==> sorted (Ficha7.insert x l)

{-
3. Defina uma propriedade prop_ins_ord_A :: Int -> Property que re-
cebe um valor a inserir numa lista, usa o gerador orderedList para gerar
uma lista de valores, e verifica que se lista continua ordenada apos insercao
deste novo valor. O gerador orderedList esta pre-definido na biblioteca
QuickCheck, e produz uma lista de valores ordenados.
-}

prop_ins_ord_A :: Int -> Property
prop_ins_ord_A x = forAll orderedList $ \xs -> sorted (Ficha7.insert x xs)

{-
Exercicio 4. Relembre o tipo de dados usado para representar Binary Search
Trees (BST) usado na ficha de gera  c  ao  de valores em QuickCheck. Considere a
seguinte fun  c  ao  que, dada uma lista, gera uma  arvore  usando o mesmo tipo de
```

```
dados:
-}

fromList :: [Int] -> BST
fromList [] = Empty
fromList (x:xs) = Node Empty x (fromList xs)

{-
sendo que esta definicao e excessivamente simples para o problema em causa.
Considere tambem a seguinte funcao que verifica se uma arvore e uma BST
valida:
-}

data BST = Empty
         | Node BST Int BST

instance Show BST where
    show = pp_BST

pp_BST = unlines . layoutTree

indent :: [String] -> [String]
indent = map ("  "++)

layoutTree Empty = []
layoutTree (Node left here right)
        = indent (layoutTree right) ++ [show here] ++ indent (layoutTree left)

isBST :: BST -> Bool
isBST Empty = True
isBST (Node l x r) = isBST l && isBST r
                     && maybeBigger (Just x) (maybeMax l)
                     && maybeBigger (maybeMin r) (Just x)
      where maybeBigger _ Nothing = True
            maybeBigger Nothing _ = True
            maybeBigger (Just x) (Just y) = x >= y
            maybeMax Empty = Nothing
            maybeMax (Node _ x Empty) = Just x
            maybeMax (Node _ _ r) = maybeMax r
            maybeMin Empty = Nothing
            maybeMin (Node Empty x _) = Just x
            maybeMin (Node l _ _) = maybeMin l

{-
1. Defina um gerador de BST a custa da funcao fromList. Defina BST como
instancia da classe Arbitrary por forma a utilizar este gerador.
-}

{-
-- este c digo funciona
instance Arbitrary BST where
    arbitrary = do
                l <- (arbitrary :: Gen [Int])
                return (fromList l)

genBST = (arbitrary :: Gen BST)

-- R: quickCheck isBST
-- Failed
-}

{-
2. Utilize a funcao quickCheck para verificar se este gerador esta a produzir
BSTs validas. Dependendo da sua implementacao do gerador, este teste
podera ou nao passar. Caso nao passe:

relembre a definicao de BST e re-escreva o gerador de forma a pro-
duzir BSTs, ainda utilizando a funcao fromList. Relembre que os
valores de uma BST se encontram ordenados, e como tal a lista
fornecida como input a funcao fromList terao de estar tambem orde-
nados, ou alternativamente, a funcao fromList deve ordenar a lista
antes de a utilizar.
-}

genOrdList lista lo hi 0 = return lista
genOrdList lista lo hi n =
```

```
                    do v <- choose (lo,hi)
                       l <- genOrdList (lista ++ [v]) (succ v) (succ hi) (n-1)
                       return $ l

{-
--funciona so esta comentado, porque a esta instancia foi redefinida
instance Arbitrary BST where
    arbitrary = do
                l <- sized $ genOrdList [] 0 10000
                return (fromList l)


genBST = (arbitrary :: Gen BST)
-}

--R: quickCheck isBST
-- Success

{-
3. Defina uma funcao balanced :: BST -> Bool que verifica se uma BST
esta devidamente balanceada. Defina uma propriedade que utilize esta
funcao para verificar se as arvores produzidas pelo gerador anteriormente
definido sao balanceadas.
-}

height :: BST -> Int
height Empty = 0
height (Node l _ r) = 1 + (max (height l) (height r))

balanced :: BST -> Bool
balanced Empty = True
balanced (Node l _ r) = abs (height l - height r) <= 1 && balanced l && balanced r

{-
(a) Com a definicao fornecida de fromList esta propriedade nunca se
verificara. Re-escreva a funcao fromList para produzir arvores bal-
anceadas.
-}

fromList' :: [Int] -> BST
fromList' = buildBalanced . sort

buildBalanced :: [Int] -> BST
buildBalanced []   = Empty
buildBalanced elts = Node (buildBalanced $ take half elts)
                          (elts !! half)
                          (buildBalanced $ drop (half+1) elts)
    where half = length elts `div` 2


{-
4. Substitua o gerador utilizado nestas propriedades, trocando a definicao
de arbitrary, pelo gerador anteriormente definido na ficha de geracao
de valores com QuickCheck. Experimente as propriedades anteriormente
definidas com o seu gerador.

-- gera apenas arvores de procura
instance Arbitrary BST where
    arbitrary = sized $ aux 0 1000
           where aux min max 0 = genVazio
                 aux min max n = frequency [(1,genVazio)
                                           ,(4,genNodo min max n)]
                 genVazio = return Empty
                 genNodo min max n = do v <- choose(min, max)
                                        l <- aux min v (n `div` 2)
                                        r <- aux v max (n `div` 2)
                                        return (Node l v r)  -}

-- gera arvores de procura balanceadas
instance Arbitrary BST where
    arbitrary = do
                l <- (arbitrary :: Gen [Int])
                return (fromList' l)

genBST = (arbitrary :: Gen BST)

```

```haskell
{-
5. Defina algumas propriedades basicas sobre arvores (nao necessariamente
ordenadas):
-}

{-
(a) Defina a funcao mirror :: BST -> BST que inverte uma arvore. De-
fina a propriedade prop_mirror que verifica se uma   arvore   e igual
apos ser espelhada duas vezes.
-}

instance Eq BST where
    Empty == Empty = True
    Empty == _     = False
    _ == Empty     = False
    Node l x r == Node l1 y r1 = (x == y) && l == l1 && r == r1

mirror :: BST -> BST
mirror Empty = Empty
mirror (Node l x r) = Node (mirror r) x (mirror l)

prop_mirror :: BST -> Bool
prop_mirror b = b == mirror(mirror b)


{-
(b) Defina a funcao incBST :: BST -> BST que incrementa todos os
valores presentes na BST (Pode utilizar a funcao foldT da ficha
QuickCheck anterior, caso a tenha definido). Defina a propriedade
prop_isBST que verifica se uma BST se mantem valida apos a aplicacao
desta funcao.
-}

incBST :: BST -> BST
incBST Empty = Empty
incBST (Node l x r) = Node (incBST l) (x+1) (incBST r)

prop_isBST :: BST -> Bool
prop_isBST bst = (isBST bst) == (isBST (incBST bst))

{-
(c) Defina a funcao height :: BST -> Int que calcula a altura de uma
BST. Defina varias propriedades prop_height_x que testam o com-
portamento desta funcao, p.e: a altura de uma arvore e maior do que
das subarvores, a altura e positiva, etc.
-}

-- height j   foi resolvida anteriormente nesta ficha

-- a altura de uma arvore e maior do que das subarvores
prop_height_1 :: BST -> Bool
prop_height_1 Empty = True
prop_height_1 (Node l x r) = c > a && c > b
                          where a = height l
                                b = height r
                                c = 1 + max a b

-- a altura e positiva
prop_height_2 :: BST -> Bool
prop_height_2 bst = height bst >= 0

{-
Exercicio 5. Observa as seguintes propriedades que descrevem o comporta-
mento de uma funcao f.
-}

prop_f1 :: [a] -> Bool
prop_f1 l1 = null (f l1 []) && null (f [] l1)

prop_f2 :: [a] -> [b] -> Bool
prop_f2 l1 l2 = length (f l1 l2) == min (length l1) (length l2)

prop_f3_1, prop_f3_2 :: (Eq a, Eq b) => [a] -> [b] -> Property
prop_f3_1 l1 l2 = length l1 < length l2 ==> l1 == map fst (f l1 l2)
prop_f3_2 l1 l2 = length l1 > length l2 ==> l2 == map snd (f l1 l2)

```

```haskell
{-
em que null :: [a] -> Bool devolve true para uma lista vazia, e false para
qualquer outra lista.
-}

{-
1. Implementa a funcao f de forma a obedecer ao comportamento descrito
pelas propriedades anteriores.
-}

f (a:as) (b:bs) = (a,b) : f as bs
f _ _ = []

{-
2. Existe uma implementacao desta funcao ja definida na biblioteca standard
do Haskell. Descubra qual e esta funcao, e implemente uma propriedade
que demonstre que a funcao f e esta funcao ja implementada sao semel-
hantes.
-}

propisZip :: (Eq a, Eq b) => [a] -> [b] -> Bool
propisZip l b = (f l b) == (zip l b)
```