Análise e Teste de Software

QuickCheck - Property Testing

Universidade do Minho

2023/2024

1 Teste de Propriedades

O teste de código consiste em fornecer *inputs* e verificar os *outputs*. O mais comum é criar diversos testes para o mesmo caso, mas variar os dados fornecidos e resultados esperados. Contudo, esse método é laborioso e não especifica por completo o código.

Uma alternativa é definir propriedades sobre o código e deixar uma ferramenta gerar os *inputs*. Estas ferramentas, das quais o QuickCheck para a linguagem Haskell foi uma das primeiras, geram um conjunto definido de dados e passam-nos às propriedades.

2 Exercícios

Os exercícios nesta ficha focam-se na definição de testes para código Haskell usando o QuickCheck[1].

1. Tendo em conta a seguinte definição de `mulL`:

   ```
   mulL :: Num a => [a] -> a
   mulL [] = 0
   mulL (h : t) = h * mulL t
   ```

   defina as seguintes propriedades para a função:

   (a) a listagem inversa (`reverse`) da lista devolve o mesmo resultado;

   (b) o resultado para uma lista com um elemento é igual a esse elemento;

   (c) o resultado é igual ao resultado da função `product` do Haskell.

   Teste as propriedades definidas para verificar se a definição está conforme a especificação.

[1] `https://hackage.haskell.org/package/QuickCheck`

2. Relembre o tipo de dados usado para representar Binary Search Trees (BST) usado na ficha de geração de valores em QuickCheck. Considere a seguinte função que, dada uma lista, gera uma àrvore usando o mesmo tipo de dados:

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

sendo que esta definição é excessivamente simples para o problema em causa.

Considere também a seguinte função que verifica se uma árvore é uma BST válida:

```
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
```

   (a) Defina um gerador de BST à custa da função `fromList`. Defina BST como instância da classe `Arbitrary` por forma a utilizar este gerador.

   (b) Utilize a função `quickCheck` para verificar se este gerador está a produzir BSTs válidas. Dependendo da sua implementação do gerador, este teste poderá ou não passar. Caso não passe:

      i. relembre a definição de BST e re-escreva o gerador de forma a produzir BSTs, ainda utilizando a função `fromList`. Relembre que os valores de uma BST se encontram ordenados, e como tal a lista fornecida como input à função `fromList` terá de estar também ordenada, ou alternativamente, a função `fromList` deve ordenar a lista antes de a utilizar.

   (c) Defina uma função `balanced :: BST -> Bool` que verifica se uma BST está devidamente balanceada. Defina uma propriedade que utilize esta função para verificar se as árvores produzidas pelo gerador anteriormente definido são balanceadas.

i. Com a definição fornecida de `fromList` esta propriedade nunca se verificará. Re-escreva a função `fromList` para produzir árvores balanceadas.

(d) Substitua o gerador utilizado nestas propriedades, trocando a definição de `arbitrary` pelo gerador anteriormente definido na ficha de geração de valores com QuickCheck. Experimente as propriedades anteriormente definidas com o seu gerador.

3. Considere a seguinte definição da função find[2]. Aqui usa-se o nome `find'` para evitar potenciais conflitos de nome com a função pré-definida.

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

que contém um erro na sua definição.

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

(b) Corrija o código e teste novamente as propriedades.

(c) (Extra) O QuickCheck consegue gerar funções aleatórias para o uso no teste de propriedades. Para fazer uso disso, é necessário importar o módulo `Test.QuickCheck.Function`. Usando isto, defina uma propriedade que verifique se a implementação aqui fornecida de `find'` tem um comportamento igual à definição de `find`.

4. Observa as seguintes propriedades que descrevem o comportamento de uma função `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)
```

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

(a) Implementa a função `f` de forma a obedecer ao comportamento descrito pelas propriedades anteriores.

[2] `http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html#v:find`

(b) Existe uma implementação desta função já definida na biblioteca standard do Haskell. Descubra qual é esta função, e implemente uma propriedade que demonstre que a função `f` e esta função já implementada são semelhantes.

5. Por vezes, comete-se o erro de se delegar o trabalho de geração à propriedade a ser testada em vez do gerador em si. Há ferramentas que permitem o uso condicional ou limitado de geradores, no entanto estas não devem ser abusadas. Pode-se limitar os valores testados através de uma précondição:

```
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 funções para verificar se uma lista está 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:insert x ys
```

(a) 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, então essa lista continua ordenada após a inserção do novo inteiro. Explique os resultados obtidos.

(b) Acrescente à propriedade definida na alínea anterior o combinador `collect`, que irá agrupar os resultados obtidos de acordo com um critério recebido como argumento. Sugestão: Agrupe os resultados de acordo com o tamanho da lista recebida como argumento da propriedade.

(c) Defina uma propriedade `prop_ins_ord_A :: Int -> Property` que recebe um valor a inserir numa lista, usa o gerador `orderedList` para gerar uma lista de valores, e verifica que se lista continua ordenada após inserção deste novo valor. O gerador `orderedList` está pré-definido na biblioteca QuickCheck, e produz uma lista de valores ordenados.