Teste de funções com QuickCheck

Princípios de Programação

Universidade de Lisboa
Faculdade de Ciências
Departamento de Informática
Licenciatura em Engenharia Informática

Vasco Thudichum Vasconcelos
Dezembro 2015*

Introdução

Grande parte do software que produzimos tem erros. A motivação para a utilização de linguagens funcionais baseia-se em observações do tipo: "é mais difícil fazer erros de programação com linguagens funcionais" e "é mais fácil raciocionar sobre programas funcionais". Mas nem os mais experientes programadores funcionais são infalíveis. Também estes, por vezes, não se apercebem das propriedades dos seus próprios programas.

Testar um programa implica compreender qual o comportamento que é aceitável em princípio, e depois tentar descobrir se o comportamento observado está de acordo com o esperado, em todo o espaço de *input*.

O teste de programas constitui de longe o método mais comum para aumentar a fiabilidade do software que produzimos. Além disso chega a constituir 50% do custo do desenvolvimento. Este custo motiva o esforço de tentar automatizar o processo de teste, tanto quanto possível.

Programas funcionais são adequados para validação automática. Aceitamos sem esforço que funções puras são mais fáceis de testar do que programas imperativos, programas que alteram o estado da computação.

Uma ferramenta de teste deve determinar se o teste passou ou não, mas um testador humano deve decidir o que significa passar um teste, de modo a que este possa ser automatizado. A ferramenta `QuickCheck` permite definir propriedades das funções a testar. As propriedades são depois validadas num grande número de casos. O `QuickCheck` também é capaz de gerar casos de teste aleatórios. Utiliza um método simples, geração aleatória, que funciona muito bem na prática.

*Pequenos ajustes em novembro 2017, de 2018 e de 2020.

Definição de propriedades

Consideremos a função que inverte uma lista recorrendo a um parâmetro de acumulação. Como a função **reverse** faz parte do **Prelude**, chamamos à nossa função **reverse'**.

```
reverse' :: [a] -> [a]
reverse' = rev []  where
  rev acc (x:xs) = rev (x:acc) acc
  rev cc []= acc
```

Pretendemos agora testar algumas propriedades da função **reverse'**. O que constitui uma propriedade interessante? Procuramos uma expressão que relacione a função **reverse'** com outras funções (ou consigo própria) e que seja válida para todas as listas finitas. Uma propriedade que relaciona **reverse'** consigo própria é a seguinte:

> *A inversa da inversa é a lista original.*

Uma propriedade que relaciona **reverse'** com a função **length** toma a seguinte forma:

> *Uma lista e a sua inversa têm sempre o mesmo comprimento.*

Com o `QuickCheck` verificamos propriedades para valores finitos, neste caso para listas finitas. Para verificar se a nossa função de inversão verifica esta propriedade, começamos por escrever o predicado `prop_reverse_length`. Notem a convenção do `QuickCheck`: os nomes das propriedades começam sempre por `prop_`.

A propriedade é verificada se a função correspondente devolver **True** para qualquer argumento. No entanto, para testar a função precisamos de decidir qual o tipo dos elementos nas listas de teste; lembrem-se que a função **reverse'** é polimórfica. Como escolhemos o tipo dos elementos das listas de teste? Acontece que a função a testar não depende do tipo de dados: basta olhar para assinatura: **reverse'** `:: [a] -> [a]`. Deste modo o tipo escolhido não afeta o resultado, desde que haja valores "suficientes" no tipo.[1] Se escolhermos o tipo **Int**, podemos restringir o tipo da função de teste. Eis uma possível definição:

```
prop_reverse_length :: [Int] -> Bool
prop_reverse_length xs = length (reverse' xs) == length
    xs
```

Agora podemos correr uns testes.

```
ghci> prop_reverse_length []
True
ghci> prop_reverse_length [8]
```

[1] Porque é que não é boa ideia usar o tipo `()` ou **Bool**?

```
True
ghci> prop_reverse_length [4,0,1,-2,9,-3]
True
```

Funciona! (para estes três casos, bem entendido). Mas este método de teste é enfadonho. Melhor seria se a máquina fizésse este trabalho por nós. Para automatizar o processo de testes usamos o módulo `QuickTest`.

```
import Test.QuickCheck
```

O `QuickCheck` gera dados de entrada aleatórios, tais como as três listas acima e passa-os para a propriedade escolhida usando a função `quickCheck`. A função recebe uma propriedade como argumento e aplica-a a um grande número de argumentos gerados aleatoriamente—100 por omissão—reportando `OK` se o resultado for **`True`** em todos os casos

```
ghci> :m Test.QuickCheck
ghci> quickCheck prop_reverse_length
+++ OK, passed 100 tests.
```

Se a propriedade falhar, a função `quickCheck` apresenta um contra exemplo. Por exemplo, se definirmos erradamente

```
reverse' :: [a] -> [a]
reverse' xs = rev xs []
  where rev (x:xs) acc = rev xs (x:acc)
        rev [] acc     = [] -- erro
```

então o `QuickCheck` pode produzir o seguinte resultado:

```
ghci> quickCheck prop_reverse_length
*** Failed! Falsifiable (after 5 tests and 4 shrinks):
[0]
```

O contra exemplo pode ser recuperado tomando a lista `[0]` como o argumento `xs` da propriedade `prop_reverse_length`. Em particular temos

```
ghci> length (reverse' [0]) == 0
ghci> length [0] == 1
```

Teste de encontro a um modelo

Consideremos agora a função de ordenação por inserção, definida deste modo:

```
sort :: Ord a => [a] -> [a]
sort = foldr insert []

insert :: Ord a => a -> [a] -> [a]
insert x []     = [x]
```

```
insert x (y:ys)
  | x <= y    = x : y : ys
  | otherwise = y : insert x ys
```

Comecemos pela função **sort**. A maneira mais económica de testar a função é recorrendo a outra função na qual confiamos, por exemplo, a função **sort** do módulo `Data.List`. Neste caso, só precisamos de dizer que as duas funções devem produzir o mesmo resultado.

```
import qualified Data.List as List
prop_sort_is_sort :: [Int] -> Bool
prop_sort_is_sort xs = sort xs == List.sort xs
```

Para correr os nossos testes chamamos o `QuickCheck`.

```
ghci> quickCheck prop_sort_is_sort
+++ OK, passed 100 tests.
```

Funções ajudantes

Mas, e se não confiarmos na implementação **List.sort**? Neste caso temos de pensar em propriedades que caracterizem as funções de ordenação. Dizemos que uma dada função é de ordenação se

> *O resultado está ordenado, e*
> *O resultado tem exactamente os mesmos elementos que a lista original.*

Para especificar a primeira propriedade definimos uma função ajudante `sorted` que verifica se uma lista está ordenada.

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

Agora é fácil de escrever a propriedade que garante a ordenação do resultado.

```
prop_sort_sorted :: [Int] -> Bool
prop_sort_sorted xs = sorted (sort xs)
```

Para a segunda propriedade precisamos também de definir uma função ajudante que decida se duas listas têm os mesmos elementos.

```
import qualified Data.List as List
sameElems :: Eq a => [a] -> [a] -> Bool
xs `sameElems` ys = xs List.\\ ys == ys List.\\ xs
```

Equipados com a função `sameElems` podemos facilmente escrever a segunda propriedade.

```
prop_sort_same_elems :: [Int] -> Bool
prop_sort_same_elems xs = sort xs `sameElems` xs
```

Para testar a função de ordenação de encontro a propriedades, corremos os dois testes.

```
ghci> quickCheck prop_sort_sorted
+++ OK, passed 100 tests.
ghci> quickCheck prop_sort_same_elems
+++ OK, passed 100 tests.
```

Propriedades condicionais

É boa prática definir e testar propriedades de muitas (ou todas) as funções envolvidas num programa, ao invés de testar apenas a função principal. O teste de cada função individual pode conduzir à descoberta de mais erros, para além de que fica mais fácil encontrar a fonte de um erro.

Pensemos então na função de inserção e imaginemos que não temos outra função à qual recorrer. As duas propriedades de interesse são as seguintes:

> *Se o argumento estiver ordenado, então também o resultado deverá estar, e*
> *Os elementos no resultado devem ser aqueles constantes nos dois argumentos da função.*

A segunda propriedade é fácil de verificar. Escrevemos a seguinte função:

```
prop_insert_same_elems :: Int -> [Int] -> Bool
prop_insert_same_elems x xs =
  insert x xs `sameElems` (x:xs)
```

A primeira é mais problemática. Não se trata de uma simples propriedade equacional, é uma propriedade condicional: só podemos esperar uma lista ordenada à saída se apresentarmos uma lista ordenada à entrada. O `QuickCheck` dispõe de um combinador de implicação `==>` para representar propriedades condicionais. Usando implicação a segunda propriedade fica assim:

```
prop_insert_sorted :: Int -> [Int] -> Property
prop_insert_sorted x xs =
  sorted xs ==> sorted (insert x xs)
```

A verificação deste tipo de propriedades funciona de um modo diferente daquele que vimos em ação até ao momento. Em vez de verificar a propriedade para 100 casos de teste aleatórios, o `QuickCheck` tenta 100 casos de teste que satisfaçam a pré-condição. Candidato que não esteja de acordo com a pré-condição (que não esteja ordenado) é mandado fora e um novo candidato é utilizado. Deste modo quando o `QuickCheck` diz que o predicado é válido para 100 testes ficamos com a certeza que todos eles passaram a pré-condição.

Como o funcionamento destes testes é diferente mudamos o tipo da propriedade de `Bool` para `Property`. O processo de teste é semelhante, mas o resultado pode ser diferente:

```
ghci> quickCheck prop_insert_sorted
*** Gave up! Passed only 76 tests; 1000 discarded tests.
```

Quando a pré-condição é raramente satisfeita, podemos acabar por gerar muitos casos sem encontrar nenhum que verifique a pré-condição. Em vez de executar indefinidamente, a função `quickCheck` gera apenas um número fixo de testes (1000 por omissão). Se a função `quickCheck` não encontrar 100 testes que passem a pré-condição nas primeiras 1000 tentativas, desiste. Ficamos no entanto a saber quantos testes verificaram a pré-condição (76 neste caso).

Monitorização de dados de teste

O operador de implicação parece resolver o problema de casos de teste inválidos. Mas podemos ter confiança que a função **`insert`** verifica as propriedades de interesse?

Podemos olhar para a distribuição dos dados de teste, alterando a propriedade do seguinte modo.

```
prop_insert_distribution :: Int -> [Int] -> Property
prop_insert_distribution x xs = sorted xs ==>
  classify (null xs) "trivial" $ sorted (insert x xs)
```

Agora podemos saber que percentagem dos testes foram triviais, isto é, que foram feitos usando uma lista vazia como argumento.

```
ghci> quickCheck prop_insert_distribution
*** Gave up! Passed only 74 tests (33% trivial).
```

A função `classify` não altera a verdade dos testes, apenas classifica-os de modo a que a função `quickCheck` produza um resultado descrevendo a classificação. Vemos neste caso que um terço dos 74 testes foram triviais.

Se um terço dos testes foram feitos com a lista vazia, quantos terão sido feitos com listas de tamanho 1? e 2? e 10? Para isso usamos a função `collect` que coleciona os vários argumentos e apresenta um histograma.

```
prop_insert_histogram :: Int -> [Int] -> Property
prop_insert_histogram x xs = sorted xs ==>
  collect (length xs) $ sorted (insert x xs)
```

Se correrrmos a nova propriedade:

```
ghci> quickCheck prop_insert_histogram
*** Gave up! Passed only 73 tests:
46% 1
30% 0
 9% 2
 6% 4
 5% 3
 1% 5
```

ficamos a saber que apenas 21% (=9+6+5+1) dos testes testaram a inserção com listas com mais de 1 elemento. Apesar de tal número constituir uma boa evidencia que a nossa função verifica a propriedade, é de preocupar que as listas vazias ou singulares dominem os testes.

A razão de tal comportamento tem obviamente a ver com a pré-condição `sorted xs` que distorce a distribuição dos dados de teste. Qualquer lista vazia ou singular está ordenada, mas apenas 50% das listas com dois elementos está ordenada, e menos de 1% das listas com 5 elementos estão ordenadas. Deste modo, casos de teste com listas grandes têm maior probabilidade de ser rejeitados pela pré-condição. Este é um risco presente de cada vez que usamos propriedades condicionais, pelo que é importante monitorar a distribuição dos casos de teste.

Se tiverem curiosidade em saber quais os testes que efectivamente estão a ser gerados usamos a função `verboseCheck`.

```
ghci> verboseCheck prop_sort_same_elems
...
Passed:
[-90,295,-263,-228,-186,-325,-272,-264,-508,455,431,-469]
...
+++ OK, passed 100 tests.
```

Finalmente, se precisarmos de mais de 100 testes, podemos incrementar o número de testes usando a função `quickCheckWith` e actualizando o registo `stdArgs`:

```
ghci> quickCheckWith stdArgs {maxSuccess = 500}
                                        prop_sort_same_elems
+++ OK, passed 500 tests.
```

Geração de dados de teste

Até aqui nunca discutimos como são gerados os dados de teste. Cada tipo de dados diferente terá uma estratégia de geração própria. De um modo geral, os dados de entrada para o `QuickCheck` têm de pertencer a um tipo instância da classe `Arbitrary`. Esta classe é definida do seguinte modo:

```
class Arbitrary a where
  arbitrary :: Gen a
```

onde `Gen a` é um tipo abstrato que representa um gerador de valores do tipo `a`. Acontece que grande parte dos tipos primitivos pertencem a esta classe. Eis alguns exemplos:

```
Arbitrary Bool
Arbitrary Char
Arbitrary Double
```

```
Arbitrary Int
Arbitrary a => Arbitrary [a]
(Arbitrary a, Arbitrary b) => Arbitrary (a, b)
```

Enquanto testámos propriedades sobre tipos primitivos nunca nos tivémos de preocupar com a questão do `Arbitrary`. O caso muda de figura quando começamos a testar propriedades de tipos de dados por nós criados. Suponhamos que pretendemos testar as funções de um módulo que implementa a gestão de trânsito numa cidade. Um dos tipos de dados contidos no módulo é o seguinte:

```
data Semaforo = Verde | Amarelo | Encarnado
              deriving (Eq, Show)
```

e uma das funções que pretendemos testar é a função que avança o semáforo:

```
seguinte :: Semaforo -> Semaforo
seguinte Verde     = Amarelo
seguinte Amarelo   = Encarnado
seguinte Encarnado = Verde
```

Antes de discutirmos quais as propriedades de interesse, temos de ser capazes de gerar cores de semáforos arbitrárias, isto é, temos de tornar o tipo `Semaforo` instância da classe `Arbitrary`. Dado que `Gen a` é um tipo abstrato, o `QuickCheck` define algumas funções primitivas para aceder à sua funcionalidade. As mais importantes são as seguintes:

```
elements :: [a] -> Gen a
choose :: Random a => (a, a) -> Gen a
oneof :: [Gen a] -> Gen a
```

A função `elements` recebe uma lista de valores e retorna um gerador de valores aleatórios retirados da lista. A função `choose` recebe um par de valores e retorna um valor aleatório no intervalo definido pelos dois valores dados. A função `oneof` recebe uma lista de geradores e escolhe aleatoriamente um deles.

Para tornar o tipo `Semaforo` instância da classe `Arbitrary` podemos escrever o seguinte código:

```
instance Arbitrary Semaforo where
  arbitrary = elements [Verde, Amarelo, Encarnado]
```

Podemos obter uma amostra dos semáforos gerados por `arbitrary` usando a função `sample`:

```
ghci> sample (arbitrary :: Gen Semaforo)
Verde
Encarnado
Amarelo
Encarnado
Verde
```

```
Amarelo
Amarelo
Amarelo
Encarnado
Verde
Amarelo
```

Uma outra possibilidade prepara 3 geradores, coloca-os numa lista, e utiliza a função `oneof` para escolher um gerador que é então utilizado para criar um semáforo aleatório. Os três geradores em que estamos interessados são: **`return`** `Verde`, **`return`** `Amarelo` e **`return`** `Encarnado`. Eis como fica o código:

```
instance Arbitrary Semaforo where
  arbitrary =
    oneof [return Verde, return Amarelo,return Encarnado]
```

A terceira alternativa é a mais versátil mas também aquela em que escrevemos mais código. Para criar luzes de semáforos geramos primeiro um valor de um tipo básico do Haskell e depois traduzimos esse valor para outro no domínio de interesse. Por exemplo, podemos gerar um número entre `1` e `3` usando a função `choose` e depois mapeamos o número numa cor. Notem que tornámos explícito o tipo `Gen Int` da expressão `choose (1, 3)`. A razão prende-se com o facto de os literais `1` e `3` terem vários tipos distintos, incluindo **`Int`** e **`Double`**.

```
instance Arbitrary Semaforo where
  arbitrary = do
    n <- choose (1, 3) :: Gen Int
    return $ case n of
      1 -> Verde
      2 -> Amarelo
      3 -> Encarnado
```

Agora já podemos testar algumas propriedades. Por exemplo:

```
prop_semaforo_circular :: Semaforo -> Bool
prop_semaforo_circular s =
  (seguinte . seguinte . seguinte) s == s
```

Correndo o `QuickCheck` ficamos a saber que os semáforos mudam de uma forma circular.

```
ghci> quickCheck prop_semaforo_circular
+++ OK, passed 100 tests.
```

Neste caso em particular, três testes manuais, bem escolhidos, teriam resolvido o problema de uma vez por todas. Quais?

Imaginemos agora que pretendemos testar um módulo para manipular documentos de texto. Um documento pode ser vazio, pode ser uma frase ou então a concatenação de dois documentos. Escrevemos o tipo deste modo:

```
data Doc = Vazio | Frase String | Concat Doc Doc
         deriving Show
```

Para gerar um documento arbitrário seguimos a seguinte estratégia. Escolhemos aleatoriamente um número entre 1 e 3. Se for 1, geramos `Vazio`. Se sair 2, geramos uma string `s` arbitrária e produzimos o documento `Frase s`. Finalmente, se sair 3, geramos recursivamente dois documentos `d1` e `d2` arbitrários e construimos o documento `Concat d1 d2`. Eis o código:

```
instance Arbitrary Doc where                                    1
  arbitrary = do                                                2
    n <- choose (1, 3) :: Gen Int                               3
    case n of                                                   4
      1 -> return Vazio                                         5
      2 -> do s <- arbitrary :: String                          6
              return $ Frase s                                  7
      3 -> do d1 <- arbitrary :: Gen Doc                        8
              d2 <- arbitrary :: Gen Doc                        9
              return $ Concat d1 d2                             10
```

Nas linhas 6, 8 e 9 anotámos a expressão `arbitrary` com o seu tipo. Tal não é absolutamente necessário, ao contrário da anotação da linha 3 que é obrigatória como vimos anteriormente.

O código é relativamente simples, mas verboso. Uma alternativa é construir 3 geradores (para documentos vazios, para frases e para concatenações) e depois usar a função `oneof`. O gerador para documentos vazios é simples: `return Vazio`. Dado um documento vazio (do tipo `Doc`) constrói um gerador de documentos vazios (do tipo `Gen Doc`).

Para gerar frases utilizamos o combinador **`liftM`** que evita nomear a string `s` no código acima. A expressão **`liftM`** `Frase arbitrary` pega na string `s` da ação `arbitrary` e constrói um documento `Frase s`. Depois constroi um gerador de frases **`return`** `(Frase s)`, exactamente como nas linhas 6–7 do código acima.

Finalmente, para gerar documentos construidos com `Concat`, utilizamos o combinador **`liftM2`** que comporta-se como **`liftM`** mas que espera dois argumentos. A função **`liftM2`** comporta-se exatamente como o codigo das linhas 8–10 do código acima.

```
import Control.Monad
instance Arbitrary Doc where
  arbitrary = oneof
    [ return Vazio,
    , liftM  Frase arbitrary,
    , liftM2 Concat arbitrary arbitrary
    ]
```

Juntemos agora duas funções de manipulação de documentos. A função `dimensao` dá o número de frases contidas num documento:

```
dimensao :: Doc -> Int
dimensao Vazio          = 0
dimensao (Frase _)      = 1
dimensao (Concat d1 d2) = dimensao d1 + dimensao d2
```

e a função `espalmar` dá uma lista com todas as frases não vazias constantes num documento:

```
espalmar :: Doc -> [String]
espalmar d = filter (not . null) (paraLista d) where
  paraLista Vazio          = []
  paraLista (Frase s)      = [s]
  paraLista (Concat d1 d2) = paraLista d1 ++ paraLista d2
```

Uma propriedade óbvia que gostariamos que fosse verdade para todos os documentos é que a dimensão de um documento deve ser igual ao comprimento da lista obtida espalmando o documento.

```
prop_dimensao_espalmar :: Doc -> Bool
prop_dimensao_espalmar d =
  dimensao d == length (espalmar d)
```

Podemos agora correr o nosso teste.

```
ghci> quickCheck prop_dimensao_espalmar
*** Failed! Falsifiable (after 4 tests):
Concat (Concat (Concat Vazio Vazio)
  (Concat (Frase "") (Frase "W"))) (Frase "\218")
```

Ooops! Algo correu mal. Conseguem descobrir o erro?

Convém sempre verificar a distribuição dos casos de teste gerados pela ação `arbitrary`. A função `collect`, introduzida na página 6, constitui uma ajuda preciosa, mas precisamos de um modo de classificar quantitativamente um documento. Uma possibilidade passa por olhar para o documento como uma árvore e contar o número de nós na árvore. A função `nos` faz exatamente isso.

```
nos :: Doc -> Int
nos Vazio = 1
nos (Frase _) = 1
nos (Concat d1 d2) = 1 + nos d1 + nos d2
```

Notem que o número de nós no documento não coincide com a sua dimensão. Para usarmos a função `collect` precisamos de uma propriedade. A propriedade mais simples é `True`. Podemos então usar o seguinte código:

```
prop_distribuicao d = collect (nos d) True


ghci> quickCheck prop_distribuicao
+++ OK, passed 100 tests:
66% 1
```

```
15% 3
 6% 9
 4% 5
 3% 7
 2% 17
 1% 29
 1% 23
 1% 21
 1% 11
```

Como seria de esperar vemos uma grande tendência para a geração de documentos com um nó. Na verdade 2/3 dos documentos gerados têm dimensão 1. O que não causa surpresas pois dos três construtores de documentos, dois têm dimensão 1: o `Vazio` e o `Frase`. Mesmo assim vemos que são gerados alguns casos de testes com um número razoável de nós. Podemos tentar contrariar esta tendência utilizando as funções `frequency` e `sized`. O artigo [1] explica como.

Para saber mais

Estas notas são baseadas no artigo [1] e no livro [2], referências estas onde poderão aprofundar os vossos conhecimentos sobre `QuickCheck`. Não se esqueçam do Hoogle.

Referências

[1] Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *Advanced Functional Programming, 4th International School*, volume 2638 of *Lecture Notes in Computer Science*, pages 59–99. Springer, 2002.

[2] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.