QuickCheck

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo adaptado a partir do material preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

QuickCheck: testes baseados em propriedades

Testando propriedades do algoritmo

UFABC

- Uma parte importante da Engenharia de Software é o teste de seu produto final. Dado que o programa compilou sem erros, ele faz o que é esperado?
- O Haskell permite, em algumas situações, provar matematicamente que seu programa está correto (usando indução).
- Outra forma de verificar a corretude é fazer testes de entrada e saída das funções criadas e verificar se elas apresentam as propriedades esperadas.

QuickCheck: introdução

UFABC

- O QuickCheck é uma ferramenta para teste de código baseado em propriedades
  - Princípios
    1. Define-se uma propriedade (invariante) do algoritmo
    2. O mecanismo de teste gera automática e aleatoriamente casos de teste
- Método complementar aos testes baseados em exemplos (i.e. testes de unidade)

Testando propriedades do algoritmo

UFABC

Se você criou um novo algoritmo de ordenação, que propriedades são esperadas?

- A lista de saída está ordenada
- A lista de saída tem o mesmo tamanho da lista de entrada
- A lista de saída contém os mesmos elementos da lista de entrada

Criando um projeto

UFABC

Vamos criar nosso primeiro projeto completo com o `stack`:

```
> stack new quickcheck simple
```

Criando um projeto

UFABC

- Edite o arquivo `quickcheck.cabal` e acrescente o seguinte ao final da linha `build-depends`:

```
build-depends: base >= 4.7 && <5, QuickCheck
```

Criando um projeto

UFABC

Digite:

```
> stack setup
> stack build
```

Exemplo 1: QuickSort

Testando QuickSort

UFABC

Uma famosa implementação no estilo funcional de QuickSort é mostrada abaixo:

- Copie e cole em seu arquivo `Main.hs`

```haskell
import Test.QuickCheck

qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
  where
    lhs = [e | e <- xs, e < x]
    rhs = [e | e <- xs, e > x]
```

Esse código contém um erro!

Funções de propriedades

UFABC

Vamos testar uma primeira propriedade de algoritmos de ordenação: idempotência.

Queremos mostrar que `qsort (qsort xs) == qsort xs`:

```haskell
prop_idempotencia :: Ord a => [a] -> Bool
prop_idempotencia xs = qsort (qsort xs) == qsort xs
```

Idempotência

UFABC

Vamos testar essa função no ghci (use `stack ghci` no diretório do seu projeto):

```
> :r
> prop_idempotencia [1]
True
> prop_idempotencia [1,2,3,4]
True
> prop_idempotencia [3,2,4,1]
True
> prop_idempotencia [4,3,2,1]
True
> prop_idempotencia []
True
```

Tamanho da lista

UFABC

Outra propriedade é que o tamanho da lista seja o mesmo após a execução do algoritmo:

```haskell
prop_length :: Ord a => [a] -> Bool
prop_length xs = length (qsort xs) == length xs
```

Tamanho da lista

UFABC

```
> :r
> prop_length [1]
True
> prop_length [1,2,3,4]
True
> prop_length [3,2,4,1]
True
> prop_length [4,3,2,1]
True
> prop_length []
True
```

Casos de teste

UFABC

- Os casos de teste utilizado são representativos?
- A biblioteca `QuickCheck` automatiza a geração de dados para testes (e faz outras coisas úteis também).

Verificando nossas propriedades

UFABC

```
> quickCheck prop_idempotencia
+++ OK, passed 100 tests.
> quickCheck prop_length
*** Failed! Falsifiable (after 4 tests):
[(),()]
```

Oops!

UFABC

- A biblioteca QuickCheck gera casos de testes progressivos, começando de casos simples até casos mais complexos em busca de erros.
- Ao encontrar um erro, ele retorna a instância mais simples que deu errado.

Para entender melhor vamos executar essa função para listas de inteiros:

```
1 > quickCheck (prop_length :: [Int] -> Bool)
2 *** Failed! Falsifiable (after 5 tests and 1 shrink):
3 [1,1]
```

- O que houve?

UFABC

```
1 > qsort [1,1]
2 [1]
```

Corrigindo o QuickSort

UFABC

Basta alterar para não descartar os elementos iguais a x:

```haskell
import Test.QuickCheck

qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
  where
    lhs = [e | e <- xs, e <= x]
    rhs = [e | e <- xs, e > x]
```

Agora sim! 🎉

```
> quickCheck (prop_length :: [Int] -> Bool)
+++ OK, passed 100 tests.
```

Quicksort com `filter`

UFABC

- Veremos funções de alta ordem nas próximas aulas.
- Se as usarmos, o código que fizemos com compreensão de listas pode ser simplificado.

```haskell
import Test.QuickCheck

qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
  where
    lhs = filter (<=x) xs
    rhs = filter (>x) xs
```

Mínimo == head

UFABC

Outra propriedade é que primeiro elemento da lista é igual ao mínimo:

```haskell
prop_minimum :: Ord a => [a] -> Bool
prop_minimum xs = head (qsort xs) == minimum xs
```

Mínimo == head

UFABC

Vamos testar essa função no ghci (use `stack ghci`):

```
1 > quickCheck prop_minimum
2 *** Failed! Exception:
3 'Prelude.head: empty list' (after 1 test):
4 []
```

Mínimo == head

UFABC

Tanto a função **minimum** quanto a função **head** retornam erro em listas vazias, podemos especificar que não queremos testar instâncias nulas com o operador **==>** (implicação):

```
1 prop_minimum :: Ord a => [a] -> Property
2 prop_minimum xs = not (null xs)
3                   ==> head (qsort xs) == minimum xs
```

Esse operador retorna uma propriedade interpretável pelo quickCheck.

Mínimo == head

UFABC

- Vamos testar essa função no ghci (use `stack ghci`):

```
1 > quickCheck prop_minimum
2 +++ OK, passed 100 tests.
```

UFABC

Finalmente, se temos um algoritmo que cumpre a mesma tarefa e temos certeza de que está correto, podemos usá-lo na comparação:

```haskell
import Data.List  -- sort

prop_model :: Ord a => [a] -> Bool
prop_model xs = qsort xs == sort xs
```

Exemplo 2: Testando a paridade

UFABC

- Queremos escrever uma função que devolva `True` caso o número recebido como parâmetro seja par, e `False` caso contrário.

```haskell
par x = x `mod` 2 == 0
```

- Para testar a função, defino a propriedade: *o sucessor de todo par é ímpar*

```haskell
prop_alternanciaParImpar :: Integral a => a -> Bool
prop_alternanciaParImpar n = par n /= par (n + 1)
```

QuickCheck - Números pares

UFABC

- Vamos testar essa função no ghci (use `stack ghci`):

```
1  > quickCheck prop_alternanciaParImpar
2  +++ OK, passed 100 tests.
```

Se par não ímpar

UFABC

- Como já temos a função `par`, é um tanto desnecessário definir a função ímpar como abaixo. Mas o exemplo é interessante para mostrar o uso da função `==>` (implicação)
- Quero testar com a propriedade: *Se n é par logo não é ímpar*

```
1  impar x = x `rem` 2 == 1
2
3  prop_seImparNaoPar n = par n ==> not (impar n)
```

Exemplo 3: Fatorial

Fatorial

UFABC

- Defina uma função que calcule o fatorial de um número
- Teste o correto funcionamento da função usando o QuickCheck

Fatorial - Tentativa 1

UFABC

- Propriedade: $(n+1)! = (n+1) \cdot n!$

```haskell
fatorial:: Integral a => a -> a
fatorial n
    | n == 1 = 1
    | otherwise = n * fatorial (n - 1)

prop_fatorialNFatorialNMaisUm n =
    fatorial n * (n + 1) == fatorial (n + 1)
```

UFABC

- Propriedade: $(n+1)! = (n+1) \cdot n!$

```haskell
fatorial:: Integral a => a -> a
fatorial n
    | n == 1 = 1
    | otherwise = n * fatorial (n - 1)

prop_fatorialNFatorialNMaisUm n =
    fatorial n * (n + 1) == fatorial (n + 1)
```

- Não funciona pois não contempla `0`!

Fatorial - Tentativa 2

UFABC

```haskell
fatorial:: Integral a => a -> a
fatorial n
    | n == 0 = 1
    | otherwise = n * fatorial (n - 1)

prop_fatorialNFatorialNMaisUm n =
    fatorial n * (n + 1) == fatorial (n + 1)
```

- E agora, vai?

Fatorial - Tentativa 2

UFABC

```haskell
fatorial:: Integral a => a -> a
fatorial n
    | n == 0 = 1
    | otherwise = n * fatorial (n - 1)

prop_fatorialNFatorialNMaisUm n =
    fatorial n * (n + 1) == fatorial (n + 1)
```

- Também não funciona: desta vez o problema está na propriedade.
- Não especificamos que só faz sentido para números positivos.
- Poderíamos resolver usando `==>` como anteriormente, mas há um jeito melhor!

Fatorial - Solução

UFABC

```haskell
fatorial:: Integral a => a -> a
fatorial n
    | n == 0 = 1
    | otherwise = n * fatorial (n - 1)

prop_fatorialNFatorialNMaisUm (NonNegative n) =
    fatorial n * (n + 1) == fatorial (n + 1)
```

- Veja http://hackage.haskell.org/package/QuickCheck-2.13.2/docs/Test-QuickCheck-Modifiers.html para uma lista dos modificadores pré-existentes
- Também é possível criar o seu próprio gerador.

QuickCheck *vs.* Conjectura de Collatz

QuickCheck Enfrenta a Conjectura de Collatz

UFABC

- A conjectura de Collatz é bem simples de formular de maneira informal:
- Dado um número natural, se o número é par então divida por 2; senão multiplique por 3 e adicione 1.
- A conjectura afirma que, após um número finito de iterações, alcança-se o número 1 para qualquer número inicial.

- Vamos utilizar o QuickCheck para verificar a conjectura
- Lembre-se o QuickCheck dizer que algo passou em X testes apenas quer dizer que ele **não encontrou** nenhum contra-exemplo para a propriedade sendo testada, não que a propriedade tenha sido provada.
- Isso é análogo a Unit Tests, onde os testes passarem não provam a ausência de bugs.

Collatz

UFABC

```haskell
collatz :: Integral a => a -> a
collatz 1 = 1
collatz n
  | par n = collatz (n `div` 2)
  | otherwise = collatz (3 * n + 1)

prop_collatz (NonNegative n) =
  collatz n == 1
```

- Vai passar?

Collatz - Tentativa 2

UFABC

```haskell
collatz :: Integral a => a -> a
collatz 1 = 1
collatz n
  | par n = collatz (n `div` 2)
  | otherwise = collatz (3 * n + 1)

prop_collatz (Positive n) =
  collatz n == 1
```

Então temos:

```
> quickCheck prop_collatz
+++ OK, passed 100 tests.
```

UFABC

- Também podemos alterar o número de testes desejados

```
1 > quickCheckWith stdArgs {maxSuccess = 5000}
  ↪  prop_collatz
2 +++ OK, passed 5000 tests.
```

Em breve veremos records e a sintaxe acima.

Para saber mais

UFABC

- An introduction to QuickCheck testing: `https://www.schoolofhaskell.com/user/pbv/an-introduction-to-quickcheck-testing`
- QuickCheck and Magic of Testing: `https://www.fpcomplete.com/blog/2017/01/quickcheck`
- Leia a documentação em: `https://hackage.haskell.org/package/QuickCheck`