Análise e Teste de Software

QuickCheck - Generators

Universidade do Minho

2023/2024

1 Geração de Valores com QuickCheck

A geração de valores consiste na instanciação de um tipo de dados, possivelmente com restrições e distribuições específicas, para um valor concreto. Estes valores podem servir para depois executar testes, e.g., testes de sistema e teste de propriedades, com um conjunto alargado de valores diferentes sem ser necessário especificar cada um individualmente.

1.1 Geração Simples

Restrições consistem em seleccionar o subconjunto de valores de um tipo de dados dentro do domínio do teste. Por exemplo, podemos definir uma nota $n$de um aluno sendo um inteiro ($n \in \mathbb{N}_0$) e restringida aos valores $n \in [0, 20]$. Isto é facilmente implementado com o QuickCheck em Haskell:

```
genNota :: Gen Word
genNota = choose (0,20)
```

Também é possível implementar a mesma funçãi escolhendo valores de uma lista:

```
genNota' :: Gen Word
genNota' = elements [0..20]
```

Estas funções retornam cada elemento com a mesma probabilidade.

Caso os valores a obter não sigam uma distribuição uniforme, é possível dar um peso a cada elemento de forma a obter uns com mais probabilidade do que outros. Por exemplo, podemos definir uma função para escolher uma marca de carro, retornando com maior probabilidade as marcas mais comuns:

```
genMarca :: Gen String
genMarca = frequency [(120,return "Renault"),(85,return "Mercedes")
                     ,(12,return "Porsche"),(4,return "Ferrari")]
```

Para experimentar um gerador, pode-se usar a função `sample` que apresenta um conjunto de valores gerados com o gerador dado. Exemplos (no interpretador):

```
> sample genNota
...
> sample genNota'
...
> sample genMarca
...
```

1.2 Exercícios

1. Escreva um gerador para automóveis cujos atributos constam dos seguintes tipos de dados:

   ```
   data Carro = Carro Tipo Marca Matricula NIF CPKm Autonomia
                deriving Show

   data Tipo  = Combustao
              | Eletrico
              | Hibrido
                deriving Show

   type Marca      = String
   type Matricula  = String
   type NIF        = String      -- NIF Proprietario
   type CPKm       = Float       -- Consumo por Km
   type Autonomia  = Int
   ```

   De acordo com as estatísticas do ACP, atualmente 70% dos automóveis usam motores a combustão, 25% são hibridos e apenas 5% são elétricos. O consumo por quilómetro (CPKm) é dado em euros e é um valor entre 0.1 e 2 euros. As marcas automóveis existentes em Portugal podem ser obtidas manualmente no site do **Standvirtual**. A matricula tem o seguinte formato "AA-11-22".

   A função que gera aleatoriamente um carro de acordo com estes requisitos recebe uma lista de NIFs válidos e tem tipo:

   ```
   genCarro :: [NIF] -> Gen Carro
   ```

   Para construir este gerador, implemente:

   (a) um gerador `genTipo` que gera um tipo de carro, de acordo com as estatísticas anteriormente fornecidas.

   (b) um gerador `genCPKm` que gera um valor de consumo por Km aleatório, de acordo com os valores anteriormente fornecidos.

(c) um gerador `genAutonomia` que gera um valor de autonomia aleatório. Para tal, faça uma pesquisa rápida para encontrar valores razoáveis para usar neste gerador.

(d) um gerador `genMarca` que gera uma marca aleatória. Para tal, use as marcas existentes em Portugal.

(e) um gerador `genMatricula` que gera uma matrícula aleatória, de acordo com o padrão anteriormente fornecido.

(f) um gerador `genCarro` que, dado uma lista de NIFs válidos, gera um carro, utilizando um desses NIFs e todos os geradores anteriormente implementados.

2. Defina um tipo de dados para representar estudantes e as suas notas. Um estudante é definido pelo seu nome, número e um tipo de registo (Normal, Militar, Trabalhador). Defina um gerador para estudantes, sabendo que 80% dos estudantes são normais, 15% são trabalhadores estudantes e 5% são militares.

3. Considere o seguinte tipo de dados para definir expressões aritméticas:

```
data Expr = Add Expr Expr
          | Mul Expr Expr
          | Const Float
            deriving Show
```

(a) Implemente um gerador para expressões aritméticas, em que 80% das operações são somas e 20% das operações são multiplicações. A probabilidade de se gerar uma expressão aritmética ou um valor constante na chamada recursiva deverá ser igual.

(b) Re-implemente este gerador para expressões aritméticas, utilizando o combinador `sized` para definir o tamanho da expressão aritmética. Usando este combinador, o gerador irá receber um valor como argumento que indicará o tamanho da expressão a gerar.

(c) Adicione um novo construtor `Var String` para representar uma variável neste tipo de dados. Adapte o seu gerador para receber uma lista de identificadores válidos, sendo que só poderão ser gerados usos de variáveis que estejam nessa lista. O novo tipo do gerador deverá ser

```
genExpr :: [String] -> Gen Expr
```

4. Considere o seguinte tipo de dados para representar Binary Search Trees (BST), em que se guardam números inteiros nos nodos da árvore:

```
data BST = Empty
         | Node BST Int BST
           deriving Show
```

(a) Define um gerador para BSTs.

(b) Defina funções sobre BSTs, e experimente as funções definidas em árvores geradas:

i. `size :: BST -> Int`
ii. `height :: BST -> Int`
iii. `max :: BST -> Int`
iv. `inorder :: BST -> [Int]`
v. `ordered :: BST -> Bool`
vi. `balanced :: BST -> Bool`
vii. `foldT :: (Int -> a -> a) -> a -> BST -> a`

5. (Extra) Considere uma linguagem *Block*, na qual apenas de pode declarar variáveis, usar variáveis, ou criar novos niveis de nesting i.e. blocks. Esta é definida pelos seguintes tipos de dados:

```
data P = R Its
type Its = [It]
data It = Block Its
        | Decl String
        | Use String
```

(a) Implemente um gerador para blocos de código nesta linguagem, sem qualquer noçãao de validade de código.

(b) Defina uma função em Haskell para fazer pretty-printing de um programa `P`. Use-a para definir `P` como instância da classe `Show`.

(c) Defina uma função em Haskell que verifique a validade de um `P`. Para ser válido, um `P` não pode ter nenhuma variável declarada duas vezes no mesmo nível, e não pode usar variáveis não declaradas. Tem também de ter atenção para apenas permitir a utilização de variáveis declaradas no mesmo nível de nesting ou em níveis acima, mas notando que uma variável pode ser declarada *depois* do seu uso, desde que num nível de nesting válido.

(d) Use a função definida anteriormente para testar o seu gerador. Altere-o para apenas gerar blocos válidos, teste-o com a função anterior e defina uma propriedade que valide que todos os blocos gerados são válidos.

6. (Extra) Considere a linguagem `Let` que modela expressões *let* em linguagens de programação como o *Haskell*. Considere também o seguinte tipo de dados para as representar:

```
data Let = Let Items Expr

type Items = [Item]
```

```
data Item = NestedLet String Let
          | Decl      String Expr
```

(a) Defina um gerador para expressões `Let`. Para isso, utilize também o gerador definido em exercícios anteriores para expressões aritméticas.

(b) Defina uma função em Haskell para fazer pretty-printing de um `Let`. Use-a para definir `Let` como instância da classe `Show`.

(c) Defina uma função em Haskell que verifique a validade de um `Let`. Para ser válido, um `Let` não pode ter nenhuma variável declarada duas vezes no mesmo nível, e não pode usar variáveis não declaradas. Tem também de ter atenção para apenas permitir a utilização de variáveis declaradas no mesmo nível de nesting ou em níveis acima, mas notando que uma variável pode ser declarada *depois* do seu uso, desde que num nível de nesting válido.

(d) Use a função definida anteriormente para testar o seu gerador. Altere-o para apenas gerar *lets* válidos, teste-o com a função anterior e defina uma propriedade que valide que todos os blocos gerados são válidos.

(e) Altere o seu gerador para utilizar o combinador *sized* para regular o tamanho dos *lets* gerados.

1.3 Geração com Estado

Gerações mais complexas podem envolver a criação de um estado para seleção correcta dos valores seguintes. Usando QuickCheck, é possível implementar isso de duas formas:

- implementar todas os geradores recebendo e alterando um estado;
- pôr o gerador numa pilha de transformers.

A primeira opção não é ideal pois não nos é possível reutilizar as funções para geração de listas de elementos (entre outros geradores). Neste caso seria necessário reimplementar essas funções.

A segunda opção é mais complexa, mas fornece muito mais possibilidades. O tipo do gerador fica:

```
type Gerador st a = StateT st Gen a
```

sendo possível executá-lo com:

```
executar :: st -> Gerador st a -> Gen a
executar st g = evalStateT g st
```

Dentro de um gerador será necessário agora fazer `lift` das funções do gerador:

```
genNotaSt :: Gerador () Word
genNotaSt = lift $ choose (0,20)
```

mas é possível aceder diretamente ao estado para ler o seu valor:

```
genNotaSt' :: Gerador (Word, Word) Word
genNotaSt' = get >>= \ (a,b) -> lift $ choose (a,b)
```

ou para guardar novo valor:

```
genNotaSt'' :: Gerador [Word] Word
genNotaSt'' = do
        l <- get
        n <- lift $ choose (a,b)
        put (n:l)
        return n
```

1.4 Exercícios

1. Implemente um gerador de números únicos, i.e., a cada geração de um número, esse número não poderá ter sido gerado antes.