Estrutura de projetos, design e testes.

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```
module Main where

import Data.Char
import Test.Tasty
import Test.Tasty.HUnit      as TH
import Test.Tasty.QuickCheck as TQ
import Test.QuickCheck
```

Objetivos

- Entender como construir aplicações usando o Haskell Stack
  - Entender o mecanismo de módulos, imports e exports.
  - Configurar dependências de um projeto
- Discussão sobre boas práticas.
- Introdução aos testes baseados em propriedadades.

Visão geral

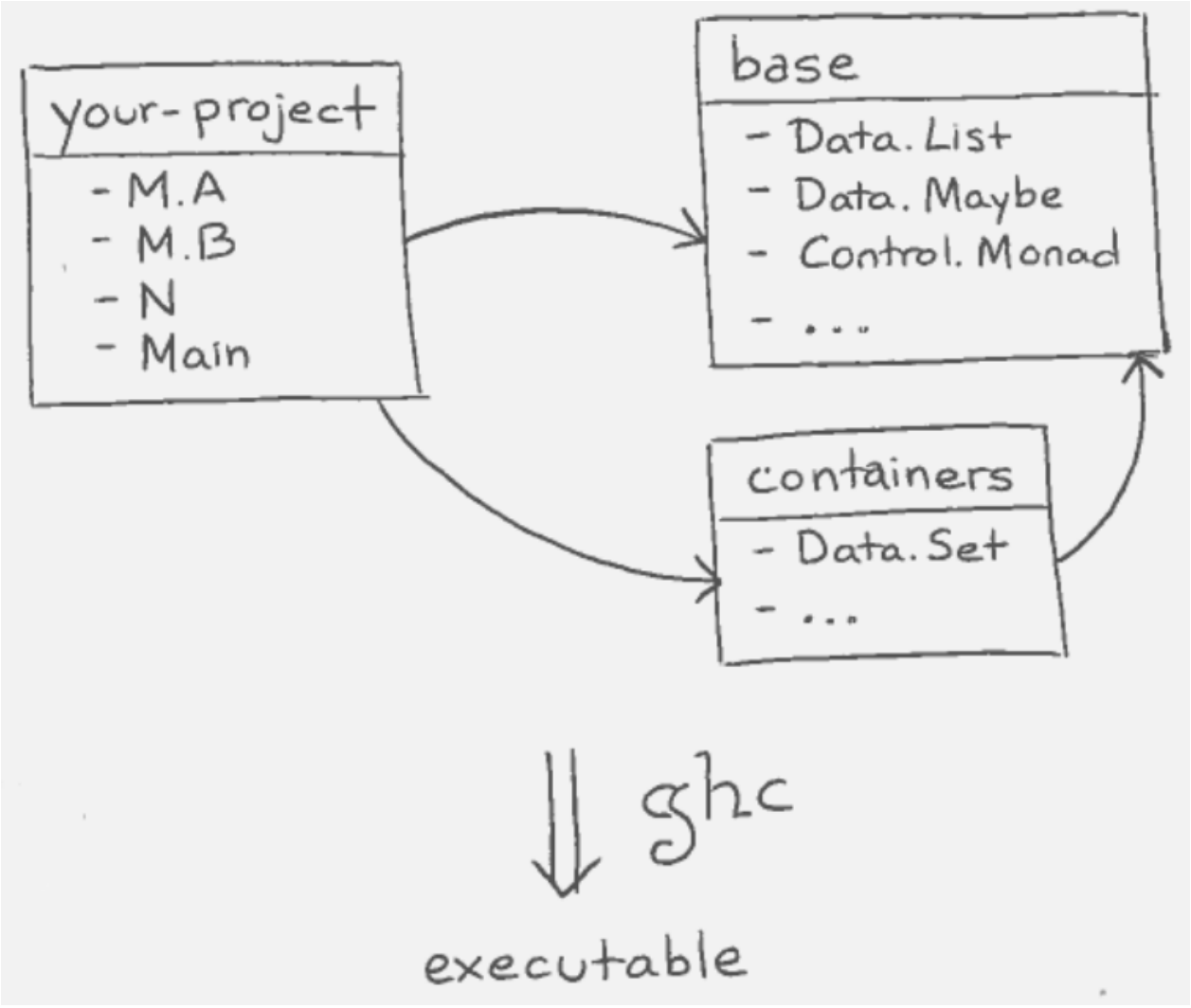

Figure 1: Estrutura geral

Pacotes e módulos

- Programas Haskell são organizados em pacotes e módulos.
- *Pacotes* são unidades de distribuição de código.
  - Seu projeto pode adicioná-los como dependências.
- Cada pacote disponibiliza um ou mais módulos.
  - Módulos definem tipos, classes e funções que podem ser exportados (i.e. public)
  - Você pode utilizar definições de outros módulos importando-os.

Estrutura de um projeto

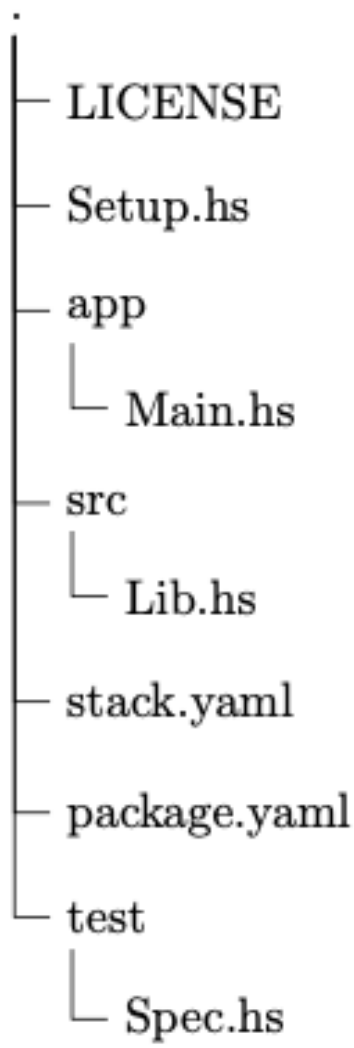

Figure 2: Estrutura geral

Estrutura de um projeto

- Arquivo package.yaml
  - Descreve dependências e outras configurações.
- Dependências

```
dependencies:
- base >= 4.7 && < 5
- tasty
- tasty-hunit
- tasty-quickcheck
```

Estrutura de um projeto

- Arquivo package.yaml
  - Descreve dependências e outras configurações.
- Definição de um executável.

```
executables:
  aula11-exe:
    main:                Main.lhs
    source-dirs:         app
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    dependencies:
    - aula11
```

Estrutura de um projeto

- Criamos um novo projeto usando o stack com o comando:

```
$> stack new my-project
```

- Instalamos as dependências e compilamos o projeto usando

```
$> stack build
```

- Executamos o código (depois de compilá-lo):

```
$> stack exec my-project
```

Módulos

- Um módulo por arquivo fonte.
- Boa prática: um "conceito" por módulo
  - Ex. Módulo Data.List possui funções envolvendo listas.
- Nome do arquivo corresponde ao nome do módulo.
  - Prefixos correspondem à estrutura de diretórios.
  - Ex. My.Long.ModuleName deve ser o arquivo

`My/Long/ModuleName.hs (ou .lhs)`

Importando código

- `import Data.List`
  - Importa toda função, tipo e classes definidas no módulo.
- `import Data.List (nub, permutations)`
  - Importa somente as definições presentes na lista.
- `import Data.List hiding (nub)`
  - Importa todas as definições *exceto* as presentes na lista.
- `import qualified Data.List as L`
  - Importa todas as definições do módulo
  - Definições devem usar o qualificador `L`.
  - Ex. `L.nub`

Exportando código

- Módulos podem especificar uma lista de definições que serão visíveis quando da importação.
- Não especificar a lista de exports: todas definições públicas.
  - Problema: Dificulta manutenção.
  - Melhor abordagem: representação de tipos "oculta", mas não temos casamento de padrão.

Mantendo invariantes

- Módulos são úteis para garantir invariantes na representação de tipos.
- Exemplo: Representando nomes

```
data Name = MkName String deriving Eq
```

- Invariante: Nomes deve ter como primeiro caractere uma letra maiúscula e demais minúsculas.

Mantendo invariantes

- Como manter esse invariante?
  - Primeiro: não expor estrutura do tipo.

```
module Name ( Name     -- exporta tipo e não construtores
            , mkName   -- mkName :: String -> Name
            , render   -- render :: Name -> String
            )
```

Mantendo invariantes

- Função `mkName` impõe o invariante.

```
mkName :: String -> Name
mkName [] = MkName []
mkName (x : xs) = MkName (x' : xs')
  where
     x' = toUpper x
     xs' = map toLower xs
```

Testes

- Testes são um componente importante de qualquer desenvolvimento de software.
- Forma mais utilizada para garantia de qualidade de software.
- Porém, testes podem mostrar somente a presença de **bugs** e não sua ausência.

Correção de programas

- Um software é correto se ele atende sua especificação.
- Veremos posteriormente na disciplina como especificar e validar funcionais.

Testes

- Forma mais comum: teste de unidade.
- Validar o valor retornado por uma função de acordo com um resultado esperado.
- Para isso, usaremos a biblioteca HUnit para construir testes unitários em Haskell

Testes

- Testando a função `mkName`:

```
render :: Name -> String
render (MkName s) = s
```

- Exemplo de teste de unidade

```
mkNameTest :: TestTree
mkNameTest
  = testCase "MkName Test" (s @?= "Maria")
  where s = render (mkName "maria")
```

Testes

- Resultado da execução do teste ao executar o comando `stack exec aula11-exe`:

```
Testes
  Testes de unidade
    MkName Test: OK

All 1 tests passed (0.00s)
```

Testes

- Testes de unidade são interessantes para uma melhor compreensão de uma tarefa.
- Porém, são limitados pela criatividade do programador.
  - No nosso exemplo, testamos apenas um caso...

Testes

- Nos testes de unidade é responsabilidade do programador relacionar a entrada para uma função e o resultado esperado.
- Como entradas são escritas manualmente pelo programador, casos problemáticos podem ser "ignorados".
- Como resolver esse dilema?

Testes

- O ideal seria automatizar a geração de casos de teste (entradas).
- Porém, como garantir que o valor retornado pela função sob teste é o esperado?
  - Descrevendo uma propriedade que caracteriza um resultado correto.

Testes

- Essa abordagem para testes é conhecida como testes baseados em propriedades.
- Disponível para Haskell pela biblioteca QuickCheck.
  - Versões disponíveis para outras linguagens.

Testes

- Ingredientes do QuickCheck
  - Propriedades: Funções que retornam valores booleanos.
  - Geradores de casos de teste: Definidos usando uma classe de tipos
- QuickCheck possui uma ampla biblioteca de funções para construir geradores de casos de teste.

Testes

- Qual propriedade especifica a correção do tipo `Name`?
  - Nomes devem iniciar com letra minúscula.

```
startsWithUpper :: String -> Bool
startsWithUpper [] = True
startsWithUpper (c : _) = isUpper c

nameCorrect :: String -> Bool
nameCorrect s = startsWithUpper (render (mkName s))
```

Testes

- Como o Quickcheck testa uma propriedade?
  - Geradores de casos de teste geram entradas aleatórias.
  - Propriedades validam se a implementação está ou não correta.
  - Caso a propriedade retorne "falso", a entrada gerada é apresentada como um contra-exemplo.
- Como executar um teste?
  - Basta executar uma propriedade usando a função `quickCheck`.

Testes

- Executando o teste obtemos:

```
quickCheck nameCorrect
*** Failed! Falsified (after 4 tests and 3 shrinks):
"1"
```

- A string “1” é apresentada como um contra-exemplo.

Testes

- Ao executarmos no interpretador:

```
$> nameCorrect "1"
False
```

- Podemos corrigir o problema por fazer o teste para strings formadas apenas por letras

Testes

- Corrigindo a propriedade:

```
implies :: Bool -> Bool -> Bool
implies x y = not x || y

nameCorrect :: String -> Bool
nameCorrect s = (all isLetter s) `implies` b
   where
     b  = startsWithUpper s'
     s' = render (mkName s)
```

Testes

- Agora executando os testes, obtemos sucesso:

```
$> quickCheck nameCorrect
+++ OK, passed 100 tests.
```

Estudo de caso

- Nesta aula, veremos como usar o QuickCheck para especificar e descobrir falhas em um algoritmo de ordenação.
- Vamos considerar o algoritmo insertion sort

Estudo de caso

- Implementação com um bug...

```
sort :: [Int] -> [Int]
sort [] = []
sort (x : xs) = insert x xs

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y : ys)
       | x <= y     = x : ys
       | otherwise = y : insert x ys
```

Estudo de caso

- Primeiro passo: devemos especificar uma propriedade satisfeita por um algoritmo de ordenação
- A propriedade deve diferenciar a ordenação de outras funções sobre listas
- A propridade não deve ser atrelada a um algoritmo específico de ordenação.

Estudo de caso

- Idealmente, um algoritmo de ordenação não deveria alterar o tamanho de uma lista.

```
sortPreservesLength :: [Int] -> Bool

sortPreservesLength xs
      = length (sort xs) == length xs
```

Estudo de caso

- Podemos executar o teste executando no interpretador:

```
$> quickCheck sortPreservesLength
*** Failed! Falsified (after 5 tests and 5 shrinks):
[0,0]
```

- Quickcheck retorna um **contra-exemplo** para a propriedade.
  - Podemos usar o contra-exemplo para descobrir a falha em nossa implementação.

Estudo de caso

- Qual equação não preserva o tamanho?

```
sort :: [Int] -> [Int]
sort [] = []
sort (x : xs) = insert x xs

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y : ys)
       | x <= y    = x : ys
       | otherwise = y : insert x ys
```

Estudo de caso

- Corrigindo...

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y : ys)
       | x <= y    = x : y : ys
       | otherwise = y : insert x ys
```

Estudo de caso

- Testes passando!

```
quickCheck sortPreservesLength
+++ OK, passed 100 tests.
```

- Mas será que isso é suficiente?

Estudo de caso

- Definição de preserves.

```
preserves :: Eq b => (a -> a) -> (a -> b) -> a -> Bool
(f `preserves` p) x = p x == p (f x)
```

- Propriedades

```
sortPreservesLength = sort `preserves` length

idPreservesLength :: [Int] -> Bool
idPreservesLength = id `preserves` length
```

Estudo de caso

- Executando o teste...

```
quickCheck idPreservesLength
+++ OK, passed 100 tests.
```

- Verificamos que `id` também preserva o tamanho de listas
  - Logo, preservar o tamanho não é uma boa propriedade...

Estudo de caso

- Que propriedade faz sentido?
- Um algoritmo de ordenação deve...
  - Ordenar uma lista!

```
sorted :: [Int] -> Bool
sorted []  = True
sorted [_] = True
sorted (x : y : ys) = x < y && sorted (y : ys)

sortEnsuresSorted :: [Int] -> Bool
sortEnsuresSorted = sorted . sort
```

Estudo de caso

- Executando o teste obtemos:

```
$> quickCheck sortEnsuresSorted
*** Failed! Falsified (after 7 tests and 7 shrinks):
[0,0,-1]
```

- Ainda não foi desta vez...

Estudo de caso

- Onde está o bug desta vez?

```
sort :: [Int] -> [Int]
sort [] = []
sort (x : xs) = insert x xs

insert :: Int -> [Int] -> [Int]
```

Estudo de caso

- Onde está o bug desta vez?
  - Não chamamos `sort` recursivamente para a cauda `xs`!

```
sort :: [Int] -> [Int]
sort [] = []
sort (x : xs) = insert x (sort xs)

insert :: Int -> [Int] -> [Int]
```

Estudo de caso

- Executando o teste...

```
$> quickCheck sortEnsuresSorted
*** Failed! Falsified (after 5 tests and 1 shrink):
[3,3]
```

- Qual o erro agora?

Estudo de caso

- Depurando...

```
$> sort [3,3]
[3,3]
$> sorted [3,3]
False
```

- O erro está na especificação de lista ordenada.

Estudo de caso

- Onde está o erro?

```
sorted :: [Int] -> Bool
sorted []  = True
sorted [_] = True
sorted (x : y : ys) = x < y && sorted (y : ys)
```

Estudo de caso

- Corrigindo...

```
sorted :: [Int] -> Bool
sorted []  = True
sorted [_] = True
sorted (x : y : ys) = x <= y && sorted (y : ys)
```

- Resultado

```
$> quickCheck sortEnsuresSorted
+++ OK, passed 100 tests.
```

Estudo de caso

- Ordenação é completamente especificada por...
  - Preservar tamanho
  - Resultado deve ser uma lista ordenada.
- Na verdade, não...

```
evilSort :: [Int] -> [Int]
evilSort xs = replicate (length xs) 1
```

- A função acima satisfaz a especificação acima...

Estudo de caso

- A especificação completa deve considerar que a lista retornada é uma permutação da original.

```
permutes :: ([Int] -> [Int]) -> [Int] -> Bool
permutes f xs =  all (flip elem xs) (f xs)

sortPermutes :: [Int] -> Bool
sortPermutes xs = sort `permutes` xs
```

- Isso é suficiente para especificar completamente e dar uma garantia da correção de nosso algoritmo.

Finalizando

- Nesta aula vimos detalhes sobre o sistema de módulos e pacotes de Haskell.
- Abordamos o problema de teste e apresentamos duas abordagens para teste: teste de unidade e o teste baseado em propriedades.

Exercícios

- Desenvolva a função

```
inRange :: Int -> Int -> [Int] -> [Int]
inRange = undefined
```

que retorna os elementos de uma lista pertencentes ao intervalo especificado pelos dois primeiros parâmetros de inRange.

Exercícios

Sua implementação deve atender os testes a seguir:

```
inRangeUnit :: TestTree
inRangeUnit
    = testCase "inRage test" $
       inRange 2 5 [1..10] @?= [2,3,4,5]
```

Evidentemente, como `inRange` não está implementada, o teste irá falhar.

Exercícios

- Descreva uma propriedade que especifique o resultado esperado por sua implementação de `inRange`.

```
inRangeProperty :: Int -> Int -> [Int] -> Bool
inRangeProperty _ _ _ = True
```

Agrupando testes

```
properties :: TestTree
properties
    = testGroup "Propriedades"
        [
          TQ.testProperty "sort preserves length"
               sortPreservesLength
        , TQ.testProperty "id preserves length"
               idPreservesLength
        , TQ.testProperty "sort ensures sorting"
               sortEnsuresSorted
        , TQ.testProperty "sort permutes input"
               sortPermutes
        , TQ.testProperty "evil sort ensures sorting"
               (sorted . evilSort)
        , TQ.testProperty "inRange specification"
               inRangeProperty
        ]
```

Agrupando testes

- Testes de unidade

```
unitTests :: TestTree
unitTests = testGroup "Testes de unidade"
                      [
                        mkNameTest
                      , inRangeUnit
                      ]
```

Agrupando testes

```
tests :: TestTree
tests = testGroup "Testes" [unitTests, properties]
```

Main

```
main :: IO ()
main = defaultMain tests
```

Listas ordenadas

```
sorted :: [Int] -> Bool
sorted []  = True
sorted [_] = True
sorted (x : y : ys) = x <= y && sorted (y : ys)
```

Insertion sort

```
sort :: [Int] -> [Int]
sort [] = []
sort (x : xs) = insert x (sort xs)

insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y : ys)
      | x <= y    = x : y : ys
      | otherwise = y : insert x ys
```