Programação Funcional (MCCC015-23)
Lista de Exercícios 5
Monoid, Foldable, Functor, Applicative

Emilio Francesquini
e.francesquini@ufabc.edu.br
Universidade Federal do ABC

14 de agosto de 2024

Nesta lista de exercícios classificamos os exercícios em três categorias que refletem o esforço relativo e o XP obtido para determinação da sua nota:

- são exercícios básicos que serão suficientes apenas para te levar a saber os rudimentos do assunto. Seu XP para determinação da sua nota final na disciplina é equivalente ao de um Charmander.
- são exercícios intermediários que exigem um pouco mais de esforço. Resolver estes exercícios vai te levar a entender um pouco melhor os conceitos e você já começará a ser capaz de utilizar estes conceitos em situações diferentes que lhe forem apresentadas. Seu XP para determinação da sua nota final na disciplina é equivalente ao de um Charmeleon.
- exercícios para Pokémon Masters. O nível de dificuldade elevado te fará a entender, de verdade, os conceitos por trás do assunto (ao contrário do Charmeleon) que apenas permite que você reproduza/adapte uma aplicação do conceito. Seu XP para determinação da sua nota final na disciplina é equivalente ao de um Charizard.

Lembrete das **Leis dos Functores**:

- Identidade: `fmap id = id`
- Composição: `fmap (f . g) == fmap f . fmap g`

Lembrete das **Leis dos Applicatives**:

- Identidade: `pure id <*> v = v`

- Homomorfismo: `pure f <*> pure x = pure (f x)`
- Intercâmbio: `u <*> pure y = pure ($ y) <*> u`
- Composição: `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

Suas implementações de functores e aplicatives precisam obededer essas leis.

Exercício 1

Considere o tipo `data Resultado = Pontuacao Int | Cola`, que representa o resultado das atividades entregues por um aluno. No final do quadrimestre, deseja-se somar a pontuação de todas as atividades entregues. No entanto, no caso de `Cola`, toda a pontuação obtida até o momento deve ser descartada, pois implica reprovação automática. Implemente uma instância de monoide para `Resultado` que modele esse comportamento.

Exercício 2

Considere o tipo `data Set a = Set [a] deriving Eq`, que deve representar um conjunto arbitrário de qualquer `a`. **Invariante:** a lista armazenada pelo construtor `Set` deve sempre conter elementos únicos e ordenados.

1. Implemente uma instância de `Show` para o set, que mostre-o conforme o seguinte exemplo: `show Set [1,2,4]` → `"{1,2,4}"` (considere a função `intercalate`[1] do `Data.List`).
2. Implemente uma função `fromList :: Ord a => [a] -> Set a` que gera um conjunto a partir de uma lista.
3. Implemente uma função `member :: Ord a => a -> Set a -> Bool` que retorna se um elemento pertence àquele conjunto
4. Implemente uma função `insert :: Ord a => a -> Set a -> Set a` que adiciona o elemento passado por parâmetro no conjunto passado por parâmetro
5. Implemente uma função `delete :: Ord a => a -> Set a -> Set a` que faz o inverso da função acima

Exercício 3

Implemente uma instância de `Monoid` para `Set a`, dado que `a` seja `Ord`, utilizando a operação de união de conjuntos

[1] `https://hackage.haskell.org/package/base-4.20.0.1/docs/Data-List.html#v:intercalate`

Exercício 4

Você está abrindo uma lanchonete diferente, pois não existe um cardápio fixo. Você apenas fornece uma lista de ingredientes possíveis, e os clientes podem combiná-los como bem entenderem. Considere os tipos `data Dieta = Vegano | Vegetariano | Tradicional`, e `data Lanche = Lanche (Set String) Int Dieta`. Dessa forma, um Lanche é composto por um conjunto de ingredientes, um preço em centavos e qual a `Dieta` adequada para aquele `Lanche`. Implemente uma instância de monoide para `Dieta`, considerando o seguinte que duas dietas são combinadas usando "denominador comum", ou seja, duas dietas diferentes resultam na menos restritiva:

- Se você colocar queijo (alimento vegetariano, mas não vegano) em um lanche vegano, ele deixa de ser vegano
- Mas colocar queijo em um lanche tradicional não faz com que ele deixe de ser tradicional

Exercício 5

Implemente uma instância de monoide para o `Lanche` conforme as seguintes regras para combinar dois `Lanche` s:

- A lista de ingredientes deve ser combinada usando união de conjuntos (pode usar sua implementação de `(<>)`)
- O preço deve ser simplesmente somado
- A `Dieta` deve seguir o `(<>)` implementado anteriormente

Exercício 6

Defina a instância de `Functor` o seguinte tipo de árvores binárias:

```haskell
data Tree a = Leaf a | Node (Tree a) a (Tree a) deriving Show
```

Exercício 7

Defina a função `arvorePossui` a que retorna `True` caso a seja um valor presente na árvore e `False` caso contrário.

Exercício 8

Defina a função `contaLetras :: Tree String -> Tree Int` que recebe uma árvore onde cada nó contém uma string e devolve uma nova árvore onde cada nó contém o comprimento das strings que continham na árvore dada como entrada.

Exercício 9

Defina uma instância de `Foldable` para `Tree`. Dica! Para definir a instância de `Foldable`, basta implementar qualquer um dentre os seguintes métodos `foldMap` ou `foldr`.

Exercício 10

Implemente a função `convertString2Int :: String -> Maybe Int`, que converte, se possível, uma string para inteiro. Você pode usar as funções em `Text.Read`.

Exercício 11

Implemente a função `nothingToZero :: Maybe Int -> Int` que dado um `Maybe Int` devolve o próprio inteiro caso presente ou 0 caso contrário.

Exercício 12

Usando a instância de `Foldable` (cuja instância você definiu nos exercícios anteriores) e o monoide `Sum` (veja `Data.Monoid` para mais informações) defina a função `frutasDaArvore :: Tree String -> Int` que recebe uma árvore de strings e devolve o número total de frutas que a árvore tem. Cada nó da árvore recebida como parâmetro possui uma string que informa a quantidade de frutas que aquele nó possui. Caso a string não corresponda a um número inteiro, ignore o nó e continue a contagem.

Exercício 13

Escreva as instâncias de `Functor` e `Applicative` para o tipo `ZipList`, no qual a função pura faz uma lista infinita de cópias do argumento, e o operador `<*>` aplica cada função argumento no valor correspondente na mesma posição.

```haskell
newtype ZipList a = Z [a] deriving Show

instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList a -> ZipList b
  fmap g (Z xs) = ..

instance Applicative ZipList where
  -- pure :: a -> ZipList a
  pure x = ..
```

Exercício 14

Dado o tipo

```
data Expr a = Var a | Val Int | Add (Expr a) (Expr a) deriving Show
```

que contém variáveis de um tipo `a`, defina instâncias para esse tipo de `Functor`, `Applicative`.

Exercício 15

Defina instâncias de `Functor`, `Applicative` para os seguintes tipos:

```
newtype Identity a = Identity a
data Pair a = Pair a a
```

Exercício 16

Run-length encoding, ou RLE para os íntimos, é um método de compactação sem perdas que é muito utilizado para comprimir dados com muitas sequencias repetitivas valores. Talvez o seu uso mais comum tenha sido para compressão de arquivos de imagem simples como ícones. Considere o ADT `data RLE a = Repeat Int a (RLE a) | End deriving (Eq, Show)`. Caso se trate de uma sequência vazia utilizamos o construtor `End` e caso se trate de uma sequência não vazia, utilizamos o construtor `Repeat` com o inteiro contendo o número de elementos repetidos do tipo `a` seguido de outro `Repeat` ou um `End` para indicar o termino da sequência. Implemente a função `rleCons :: Eq a => a -> RLE a -> RLE a` cujo comportamento é análogo à função cons para listas `(:)`.

Exercício 17

Implemente uma instância de `Foldable` para `RLE`.

Exercício 18

Implemente a função `encode :: Eq a => [a] -> RLE a` que faz as conversões entre listas comuns e `RLE`. Use a instância de foldable definida no exercício anterior. Exemplos:

```
> encode [1,1,2,2,2]
   Repeat 2 1 (Repeat 3 2 End)

> encode [1,2,3,3,3,2,2]
   Repeat 1 1 (Repeat 1 2 (Repeat 3 3 (Repeat 2 2 End)))

> encode ['a','x','x','z','x']
   Repeat 1 'a' (Repeat 2 'x' (Repeat 1 'z' (Repeat 1 'x' End)))
```

Exercício 19

Implemente a função `decode :: RLE a -> [a]` que faz o processo inverso de `encode`. Use a instância de foldable definida para o tipo `RLE`.

 Exercício 20

Obedecendo as leis descritas no início da lista de exercícios, escreva instâncias de `Functor` e `Applicative` para o tipo `data Fantasma a = Fantasma`.

 Exercício 21

Obedecendo as leis descritas acima, escreva instâncias de `Functor` e `Applicative` para o tipo `data Duo a = Duo (Bool -> a)`.

 Exercício 22

Suponha que você está cursando a disciplina de Arquitetura de Computadores e precisa implementar uma hierarquia de memória para o seu emulador MIPS. Você desenvolveu a seguinte estrutura de dados:

```
data Memory a = UnifiedCache a (Memory a) | SplitCache a a (Memory a) | RAM a
```

Essa estrutura tem a propriedade interessante de que **sempre** vai terminar na memória `RAM`, e podemos ter quantos níveis de cache quisermos. Por exemplo, se quisermos armazenar os acessos a cada um dos níveis de memória, podemos fazer: `SplitCache 5 2 (UnifiedCache 10 (RAM 5))`.

Imagine que queremos calcular o total de acessos feitos a todos os níveis de memória usando:

```
foldMap Sum $ SplitCache 5 2 (UnifiedCache 10 (RAM 5))
```

Implemente instâncias de `Functor`, `Foldable` e `Traversable` para a estrutura `Memory`.