Eduardo Ribeiro de Oliveira 11201810478

Consulta de Temperatura com Haskell e Mônadas

Introdução

As mônadas têm um papel fundamental em Haskell no quesito de ações encadeadas e ações envolvendo IO. Elas permitem que operações sequenciais sejam expressas de forma clara e concisa, lidando com efeitos colaterais de maneira controlada. No caso do IO, as mônadas possibilitam a execução de ações que interagem com o ambiente externo, como fazer requisições HTTP ou ler dados de um arquivo, encapsulando essas operações em uma estrutura que garante segurança e previsibilidade.
Neste tutorial, utilizaremos do conceito de mônadas e também das mônadas IO, Async e Parser, para fazer requisições HTTP à API do OpenWeatherMap, no intuito de consultar a temperatura de uma certa cidade, e a partir da resposta, em formato JSON, sermos capazes de decodificar e conseguir a informação relevante para o nosso exemplo. Detalhando a função de cada mônada temos que a mônada IO é responsável para lidar com operações de entrada/saída, permitindo que funções executem ações que interajam com o ambiente externo de forma segura e controlada. A mônada Async, proveniente do pacote async, permite que operações sejam executadas de forma concorrente e assíncrona, neste tutorial, ela será utilizada para fazer requisições HTTP de forma paralela. Por fim, a mônada é utilizada para a desserialização de dados, neste caso, será usada para manipulação de dados no formato JSON.

Preparativos

Primeiramente, para começar, precisamos instalar os pacotes que serão utilizados neste tutorial. Portanto, precisamos utilizar os seguintes pacotes: HTTP-Simple, Aeson e Async. Para instalá-los, você pode simplesmente colar no seu terminal:

```
cabal update
cabal install async aeson HTTP-Simple
```

Após a conclusão da instalação, podemos começar nosso módulo e já importar os pacotes instalados:

```
import Control.Concurrent.Async (async, wait)
import Network.HTTP.Simple (httpLBS, getResponseBody, parseRequest)
import GHC.Generics
import Data.Aeson
import Data.Text (Text)
```

Uma breve explicação sobre a função de cada módulo:

- Async: usaremos as funções async e wait desse módulo para criar tarefas assíncronas de forma concorrente
- HTTP-Simple: módulo para auxiliar a realização das requisições HTTP
- Generics: módulo que permite derivar classes do tipo genéricas, neste caso usaremos para derivar FromJson
- Aeson: módulo para serialização e desserialização de JSON
- Text: módulo para auxílio na manipulação de texto

Implementação

Agora que lidamos com todos os pré-requisitos, podemos começar a implementação, primeiramente vamos criar um tipo de dados chamado Weather para representar uma cidade, esse TDD neste tutorial vai ter apenas dois campos: cityName, do tipo Text e temperature, do tipo Double.

```
data Weather = Weather
{ cityName :: Text
, temperature :: Double
} deriving (Show, Generic)
```

Como vamos receber da API um JSON, precisamos implementar como será feita a desserialização, para isso, precisamos analisar o JSON para pegar os valores corretamente. A implementação fica assim:

```haskell
instance FromJSON Weather where
 parseJSON = withObject "Weather" $ \obj -> do
   cityName <- obj .: "name"
   main <- obj .: "main"
   temperature <- main .: "temp"
   return Weather
     { cityName = cityName
     , temperature = temperature
     }
```

Nesse caso, usamos a mônada parse do módulo Aeson para, após as operações monádicas, contidas no bloco do, poder fazer o parsing do JSON, extrair as informações necessárias para a aplicação e retornar um valor do tipo Weather.
Agora que já implementamos o tipo de dados e também como fazer o parsing do JSON, podemos finalmente implementar como será feito o request para a API. Portanto, utilizando o módulo HTTP-Simple, podemos criar uma função que recebe uma URL e devolve um IO Weather, porque como estamos trabalhando com a API, precisamos trabalhar dentro do escopo IO. Assim fica a implementação:

```haskell
fetchTemperature :: String -> IO Weather
fetchTemperature url = do
 request <- parseRequest url
 response <- httpLBS request
 case decode (getResponseBody response) of
   Just weather -> return weather
   Nothing -> error "Falha ao tentar decodificar os dados"
```

Nessa função, trabalharemos novamente com uma mônada, nesse caso a mônada IO. A função recebe uma URL, do tipo String, e usa a função parseRequest para converter a URL em uma requisição HTTP. Então, usamos a função httpLBS para fazer a requisição e atribuir a resposta na variável response. Utilizando as funções getResponseBody e decode para, respectivamente, extrair o corpo da resposta HTTP e decodificar a resposta como valor JSON, caso a decodificação tenha tido sucesso, a função retorna o valor do tipo Weather, caso contrário, ou seja, houve um erro na decodificação, é apresentado uma mensagem de erro.
Antes de implementarmos a função main que vai ser responsável por chamar essas funções implementadas, precisamos de uma função simples porém indispensável que é a função de print do tipo de dados Weather:

```haskell
printWeather :: Weather -> IO ()
printWeather (Weather cityName temp) =
 putStrLn $ "Temperatura em " ++ show cityName ++ ": " ++ printf "%.2f" (temp -
273.15) ++ "°C"
```

Essa função simplesmente converte de Kelvin para Celsius e imprime no terminal.
Agora, finalmente podemos implementar nossa função main e começar a testar. Não resta muito o que fazer na main, apenas definir a cidade que queremos consultar e também definir a URL da API, ficando assim:

```
main :: IO ()
main = do
 let city = "Rio de Janeiro"
 let apiUrl = "https://api.openweathermap.org/data/2.5/weather?appid="++
getAPIKey ++"&q=" ++ city

 response <- fetchTemperature apiUrl
  printWeather response
```

Com tudo feito, vamos aos testes. Primeiro vamos testar com a cidade que está já no código, ou seja, uma cidade existente:

```
ghci> main
Temperatura em "Rio de Janeiro": 24.30°C
```

Perfeito! Funcionando corretamente, agora e se colocarmos uma cidade que não existe, por exemplo city ser igual a “teste”?

```
ghci> main
*** Exception: Falha ao tentar decodificar os dados
CallStack (from HasCallStack):
  error, called at tutorial1_2.hs:49:16 in main:Main
```

Temos o erro na hora de decodificar, tudo conforme o esperado. Porém, podemos ir além, podemos ter uma lista de cidades que queremos consultar a temperatura, como fazemos? Poderíamos fazer de maneira sequencial porém o tempo seria um empecilho, visto que cada requisição só iniciaria após a anterior ser impressa no terminal, no entanto, podemos fazer uso do módulo async que permite criação de tarefas em paralelo para fazer requisições para a API de forma concorrente. A implementação fica assim:

```haskell
main :: IO ()
main = do
 let cities = ["Rio de Janeiro", "London", "New York", "Tokyo"]
 let apiUrls = map (\city ->
"https://api.openweathermap.org/data/2.5/weather?appid="++ getAPIKey ++"&q=" ++
city) cities

 asyncTasks <- mapM (async . fetchTemperature) apiUrls
 responses <- mapM wait asyncTasks

 mapM_ printWeather responses
```

Nesse caso, usamos async para criar uma requisição assíncrona de cada URL da lista apiUrls, então utilizamos wait para esperar que todas as requisições sejam finalizadas para por fim, imprimir a temperatura de cada cidade da lista. O resultado fica assim:

```
ghci> main
Temperatura em "Rio de Janeiro": 24.98°C
Temperatura em "London": 13.69°C
Temperatura em "New York": 25.22°C
Temperatura em "Tokyo": 19.21°C
```

Conclusão

Neste tutorial, utilizamos as mônadas IO, Async e Parser para implementar um sistema de consulta à API do OpenWeatherMap, permitindo que façamos requisições HTTP para obter informações de temperatura de diversas cidades.

Código usado: https://github.com/eduribeiro8/DGPT