Paradigmas de Programação

Fabrício Olivetti de França

09 de Agosto de 2018

Haskell Concorrente

Programação Concorrente

O objetivo da programação concorrente é definir explicitamente múltiplos caminhos de controle, geralmente para permitir múltiplas interações com usuário ou interfaces externas.

Não necessariamente serão executadas em paralelo.

A ideia é descrever cada interação em separado, mas fazer com que elas ocorram ao mesmo tempo (intercaladamente).

Considere um Web Browser que permite carregar múltiplas páginas ao mesmo tempo.

Além disso, enquanto ele carrega uma ou mais páginas, a interface ainda interage com o usuário (apertar o botão *cancelar*, etc.)

Um servidor Web também implementa concorrência para servir múltiplos pedidos ao mesmo tempo.

Imagine a situação de um servidor Web atendendo um pedido de página por vez.

Haskell Concorrente

O Haskell fornece um conjunto de funcionalidades simples mas genéricas que podem ser utilizadas para definir estruturas mais complexas.

Com isso fica a cargo do programador definir o melhor modelo de programação concorrente.

Conceitos Básicos

Criando uma *thread*

Uma **thread** é a menor sequência de computação que pode ser gerenciada de forma independente pelo gerenciador de tarefas.

Um **processo** é um procedimento computacional completo que pode conter diversas *threads* de execução.

Criando uma *thread*

Múltiplas *threads* de um mesmo processo compartilham a memória alocada, podendo utilizá-la para troca de mensagens.

Múltiplos processos **não** compartilham memória alocada.

Criando uma *thread*

No Haskell criamos uma thread com a função `forkIO`:

```
import Control.Concurrent

forkIO :: IO () -> IO ThreadId
```

Ela recebe uma ação computacional de IO como argumento e retorna um identificador dessa *thread*.

Criando uma *thread*

A ideia é que todo efeito colateral feito pela ação IO será feito de forma concorrente com outras *threads*:

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  forkIO (replicateM_ 100000 (putChar 'A'))
  replicateM_ 100000 (putChar 'B')
```

Criando uma *thread*

A primeira linha da função `main` desativa o buffer para executar toda ação IO no momento exato em que ela é enviada.

```haskell
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  forkIO (replicateM_ 100000 (putChar 'A'))
  replicateM_ 100000 (putChar 'B')
```

Criando uma *thread*

A segunda linha cria uma *thread* que imprimirá o caractere *A* dez mil vezes.

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  forkIO (replicateM_ 100000 (putChar 'A'))
  replicateM_ 100000 (putChar 'B')
```

Criando uma *thread*

A terceira linha imprime o caractere *B* dez mil vezes.

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  forkIO (replicateM_ 100000 (putChar 'A'))
  replicateM_ 100000 (putChar 'B')
```

Criando uma *thread*

A execução do programa resultará em caracteres *A* e *B* intercalados:

AAAAAAAAABABABABABABABABABABABABAABABABABAB

...

Exemplo 1: Lembretes

Vamos criar uma função *multithread* que aguarda o usuário entrar com um tempo em segundos e cria uma thread para imprimir uma mensagem após esse número de segundos tenha passado:

```
import Control.Concurrent
import Control.Monad

main = forever $ do
           s <- getLine
           forkIO $ setReminder s
```

Exemplo 1: Lembretes

A função `forever` está definida na biblioteca `Control.Monad` e simplesmente repete a ação eternamente. A função `setReminder` pode ser definida como:

```
setReminder :: String -> IO ()
setReminder s = do
  let t = read s :: Int
  putStrLn ("Ok, adicionado para " ++ show t ++ " segs.")
  threadDelay (10^6 * t)
  putStrLn (show t ++  " segundos se passaram! BING!"
```

A função `threadDelay` suspende a execução da *thread* por *n* microsegundos.

Exemplo 1: Lembretes

Exemplo de execução:

```
$ ./lembrete
2
Ok, adicionado para 2 segs.
4
Ok, adicionado para 4 segs.
2 segundos se passaram! BING!
4 segundos se passaram! BING!
```

Exercício 01 (0.5 pto)

Altere o programa anterior para terminar o programa assim que o usuário digitar *quit*.

DICA: troque o uso de `forever` por uma função definida por você denominada `repita`.

Comunicação entre *threads*: *MVar*

Para que as *threads* possam se comunicar entre si é necessário a existência de um espaço de memória compartilhada.

No Haskell isso é implementado através do tipo `MVar`:

```
data MVar a

newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
```

MVar

O tipo `MVar` é um *container* para qualquer tipo de dado. Você pode armazenar um valor `Integer`, uma `String`, uma lista de `Bool`, etc.

A função `newEmptyMVar` cria um `MVar` inicialmente vazio. A função `newMVar` recebe um argumento `x` e retorna um `MVar` contendo `x`.

As funções `takeMVar` e `putMVar`, inserem e removem um conteúdo em um `MVar`.

MVar

Notem que `MVar` armazena apenas um único valor em sua estrutura:

- Se uma *thread* chamar `takeMVar` e ela estiver vazia, ficará bloqueada em espera até que algum conteúdo seja inserido.
- Se uma *thread* chamar `putMVar` e ela estiver cheia, ficará bloqueada em espera até que alguema *thread* utilize `takeMVar`.

Considere o seguinte código exemplo:

```
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r1 <- takeMVar m
  print r1
  r2 <- takeMVar m
  print r2
```

MVar

Inicialmente, cria-se uma `MVar` vazia:

```
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r1 <- takeMVar m
  print r1
  r2 <- takeMVar m
  print r2
```

Em seguida, criamos uma *thread* que armazena dois caracteres na sequência, ao inserir o primeiro caractere a *thread* fica bloqueada aguardando espaço ser liberado.

```
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r1 <- takeMVar m
  print r1
  r2 <- takeMVar m
  print r2
```

Nesse momento o *thread* principal recupera o valor de *MVar* e armazena em `r1`, liberando espaço para a *thread* armazenar o segundo caractere.

```
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r1 <- takeMVar m
  print r1
  r2 <- takeMVar m
  print r2
```

Ao final, o segundo caractere é recuperado e `MVar` se torna vazio.

```
main = do
  m <- newEmptyMVar
  forkIO $ do putMVar m 'x'; putMVar m 'y'
  r1 <- takeMVar m
  print r1
  r2 <- takeMVar m
  print r2
```

Mecanismo anti-burrice

E se o programador escrever o seguinte programa:

```
main = do
  m <- newEmptyMVar
  takeMVar m
```

Mecanismo anti-burrice

A execução retornará:

```
$ ./burro
burro: thread blocked indefinitely in an MVar operation
```

Se você criar um programa em que a *thread* fica bloqueada eternamente, em muitos casos o programa emite uma exceção `BlockedIndefinitelyOnMVar`.

Construindo tipos mutáveis com *MVar*

Tipos mutáveis

Como sabemos, os tipos do Haskell são imutáveis, ou seja, uma vez que definimos uma variável ela não pode mudar de valor.

O tipo *MVar* pode nos ajudar a simular um tipo mutável a partir de um tipo imutável de forma transparente!

Tipo *Map*

A biblioteca `Data.Map` fornece o tipo **mapa associativo**:

```
data Map k a
```

```
import qualified Data.Map as M
```

que define um mapa associativo com chave do tipo `k` e valores do tipo `a`.

Tipo *Map*

Essa biblioteca possui as funções:

```
M.empty  :: Map k a

M.insert :: Ord k => k -> a -> Map k a -> Map k a
M.insert k v m = -- insere valor v na chave k do mapa m,
                 -- substituindo caso já exista

M.lookup :: Ord k => k -> Map k a -> Maybe a
M.lookup k m = -- retorna o valor na chave k do mapa m,
               -- retorna Nothing caso não exista a chave
```

Agenda telefônica

Vamos criar o tipo agenda telefônica:

```
type Nome   = String
type Numero = String
type Agenda = M.Map Nome Numero
```

Agenda telefônica

Mas uma agenda telefônica não pode ser uma estrutura imutável. Preciso ter a capacidade de inserir novos nomes e atualizar as entradas. Vamos definir uma agenda telefônica mutável como:

```
newtype AgendaMut = AgendaMut (MVar Agenda)
```

Exercício (0.5 pto)

Defina a função `novaAgenda` que cria uma agenda mutável vazia.

```
novaAgenda :: IO AgendaMut
```

Exercício

Defina agora a função `insere` que insere um nome e um telefone:

```
insere :: AgendaMut -> Nome -> Numero -> IO ()
insere (AgendaMut m) nome numero = ??
```

Exercício

Defina a função `procura` que retorna uma entrada da agenda:

```
procura :: AgendaMut -> Nome -> IO (Maybe Numero)
procura (AgendaMut m) -> Nome = do
```

Agenda Mutável

Dessa forma, podemos trabalhar com a agenda da seguinte forma:

```
nomes = [("Joao", "111-222"), ("Maria", "222-111"),
         ("Marcos", "333-222")]

main = do
  s <- novaAgenda
  mapM_ (uncurry (insere s)) nomes
  print =<< procura s "Marcos"
  print =<< procura s "Ana"
```

Agenda Mutável

O operador =<< é igual ao operador >>= mas invertido. Converta a expressão:

```
print =<< procura s "Marcos"
```

para a notação do

Agenda Mutável

Se essas funções forem utilizadas em um programa que cria múltiplas *threads* podemos observar algumas vantagens:

- Durante a operação de busca não é necessário criar um *lock* no estado da agenda durante a operação.
- Graças a avaliação preguiçosa, também não se faz necessário um *lock* por muito tempo no momento da inserção.

Operações Assíncronas

Operações Assíncronas

Imagine a situação em que desejamos capturar o conteúdo de uma lista de páginas da Web. Queremos fazer isso de forma concorrente e, após o encerramento de **todas** as threads, quero aplicar alguma função nos resultados.

Operações Assíncronas (1/3)

Nosso código seria algo como:

```
import Control.Concurrent
import Data.ByteString as B
import GetURL

main = do
  m1 <- newEmptyMVar
  m2 <- newEmptyMVar
```

Operações Assíncronas (2/3)

```
forkIO $ do
  r <- getURL "http://www.wikipedia.org/wiki/Shovel"
  putMVar m1 r

forkIO $ do
  r <- getURL "http://www.wikipedia.org/wiki/Spade"
  putMVar m2 r
```

Operações Assíncronas (3/3)

```
r1 <- takeMVar m1
r2 <- takeMVar m2
print (B.length r1, B.length r2)
```

Operações Assíncronas

A ideia é que as duas *threads* façam o download do conteúdo de cada URL em *background* assincronamente.

Uma **operação assíncrona** é uma operação que é feita em *background* enquanto eu posso fazer outras operações que não dependam dela, e permita que eu aguarde o final para realizar alguma operação sobre os resultaos.

Operações Assíncronas

O código está muito repetitivo. E se eu quiser fazer a mesma operação para uma lista de URLs? Vamos generalizar:

```haskell
data Async a = Async (MVar a)
```

Operações Assíncronas

```haskell
async :: IO a -> IO (Async a)
async action = do
  var <- newEmptyMVar
  forkIO (do r <- action; putMVar var r)
  return (Async var)
```

Operações Assíncronas

```haskell
wait :: Async a -> IO a
wait (Async var) = readMVar var
```

Essas funções estão definidas na biblioteca

Control.Concurrent.Async

Operações Assíncronas

A função `async` cria uma *thread* a ser executada assíncronamente com outras.

A função `wait` aguarda o final da execução de uma *thread* assíncrona.

Operações Assíncronas

Dessa forma nosso código pode ser reescrito como:

```
import Control.Concurrent
import Data.ByteString as B
import GetURL

url1 = "http://www.wikipedia.org/wiki/Shovel"
url2 = "http://www.wikipedia.org/wiki/Spade"

main = do
  a1 <- async (getURL url1)
  a2 <- async (getURL url2)
  r1 <- wait a1
  r2 <- wait a2
  print (B.length r1, B.length r2)
```

Operações Assíncronas

Assim, podemos capturar uma lista de sites da seguinte forma:

```
main = do
  as <- mapM (async . getURL) sites
  rs <- mapM wait as
  mapM_ (r -> print $ B.length r) rs
```

Tratamento de Erro

E se uma das URLs não existir ou retornar algum erro? Devemos jogar fora todos os resultados e exibir uma mensagem de erro?

A biblioteca `async` define a função:

```
waitCatch :: Async a -> IO (Either SomeException a)
```

Que retorna ou um erro, que pode ser tratado, ou o valor esperado.
Para tratar o erro devemos importar também **import Control.Exception**

Either

O tipo `Either` é definido como:

```haskell
data Either a b = Left a | Right b
```

e assim como o **Maybe** é utilizado para tratamento de erro. Ele diz: "ou vou retornar algo do tipo a ou do tipo b", sendo o tipo *a* geralmente tratado como o erro.

Tratamento de Erro

Então podemos definir:

```
printLen :: (Either SomeException B.ByteString) -> IO ()
printLen (Left  e) = print "URL not found"
printLen (Right b) = print $ B.length b
```

Tratamento de Erro

E então:

```
main = do
  as <- mapM (async . getURL) sites
  rs <- mapM waitCatch as
  mapM_ printLen rs
```

Leituras complementares

Com isso finalizamos o assunto de Programação Concorrente nessa disciplina, embora não tenhamos esgotado todos os conceitos.

Para quem quiser avançar no assunto, a leitura do livro do Simon Marlow é obrigatória!