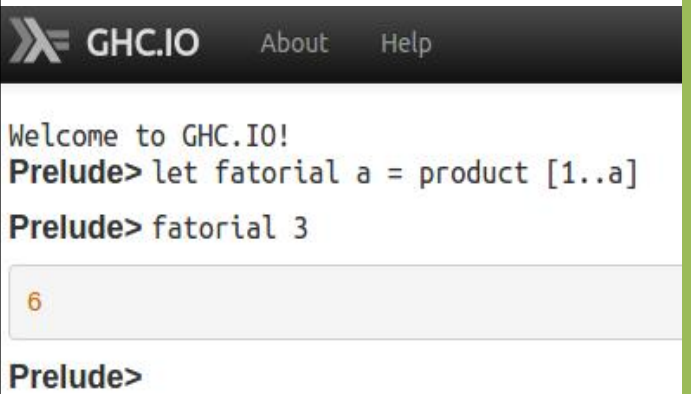

sumList [2,3,4,5]
= 2 + sumList [3,4,5]
= 2 + (3 + sumList [4,5])
= 2 + (3 + (4 + sumList [5]))
= 2 + (3 + (4 + (5 + sumList [])))
= 2 + (3 + (4 + (5 + 0)))
= 14

Programação Funcional

Tratamento de Exceções

Matheus Carvalho Viana
matheuscviana@ufsj.edu.br

Dcomp
Departamento de Ciência da Computação

UFSJ
Universidade Federal de São João del-Rei

Introdução

Bons programas devem estar preparados para reagir de maneira adequada quando ocorrem situações anômalas.

Estas situações indesejadas, conhecidas como **exceções**, podem ter várias causas:

tentativa de divisão por zero;

cálculo de raiz quadrada de número negativo;

aplicação da função fatorial a um número negativo;

tentativa de encontrar a cabeça ou a cauda de uma lista vazia;

entre outros.

Tratamento de Exceção

Uma exceção é um evento errôneo, detectável por hardware ou por software, que pode causar uma interrupção abrupta do programa

Para impedir essa interrupção, é necessário que o programa possua uma forma de **tratamento de exceção**.

Quando uma exceção ocorre, a execução do programa é transferida para um código conhecido como **tratador de exceção**, que, quando possível, mantém o programa funcionado, ou, pelo menos, alerta o usuário da ocorrência da exceção.

A natureza da exceção é que vai determinar se o tratador de exceção vai ser capaz ou não contornar o problema e manter o programa funcionando.

Tratamento de Exceção

Uma exceção é **levantada** (*raised*) quando seu evento associado ocorre.

Em Haskell, três técnicas podem ser utilizadas para realizar o tratamento de exceções:

1. Exibir um mensagem por meio de uma função de erro;
2. Indicar um valor fictício;
3. Usar o tipo Maybe a.

Função error

A solução mais simples para tratar uma exceção é exibir uma mensagem informando o tipo da exceção e parar a execução do programa.

Isso pode ser feito chamando a função **`error`**, que é fornecida pelo Prelude de Haskell.

Essa função recebe uma `String` e devolve um tipo `t` qualquer.

```
error :: String -> t
```

Função error

```
areacirculo :: Float -> Float
areacirculo r
  | r >= 0 = pi * r ^ 2
  | otherwise = error "Círculo com raio negativo"


divisao :: Int -> Int -> Int
divisao _ 0 = error "Divisão por zero"
divisao a b = if a >= b then 1 + divisao (a-b) b else 0
```

Valores Fictícios

O problema em usar a função **`error`** é que todas as informações calculadas até o ponto em que a exceção ocorreu são perdidas porque o programa é abortado.

Uma alternativa é utilizar valores fictícios nos casos inválidos de uma função, impedindo assim que o programa aborte sua execução.

Valores Fictícios

Por exemplo, a função **`tail`** é construída para retornar a cauda de uma lista finita não vazia. Mas caso a lista seja vazia, a função reporta esta situação com uma mensagem de erro e parar a execução do programa.

```
tail :: [a] -> [a]
tail (_:x) = x
tail [ ] = error "cauda de lista vazia"
```

Valores Fictícios

No entanto, a função **`tail`** poderia ser refeita da seguinte forma:

```
tail :: [a] -> [a]
tail (_:x) = x
tail [ ] = [ ]
```

Desta forma, todas as listas passam a ter uma resposta para uma solicitação de sua cauda, seja ela vazia ou não.

Se a lista não for vazia, sua cauda será reportada normalmente. Se a lista for vazia, o resultado será o valor fictício **`[]`**.

Valores Fictícios

De forma similar, a função de divisão de dois números inteiros pode ser feita da seguinte forma:

```
divisao :: Int -> Int -> Int
divisao _ 0 = 0  --valor fictício
divisao a b = if a >= b then 1 + divisao (a-b) b else 0
```

Valores Fictícios

Para as funções **`tail`** e **`divisao`**, a escolha dos valores fictícios foi simples. No entanto, existem casos em que esta escolha é complicada ou nem mesmo possível.

Por exemplo, na definição de uma função que retorne a cabeça de uma lista. Qual deve ser o valor fictício a ser adotado neste caso?

Para situações como essa, o ideal é criar uma versão da função original com um parâmetro *default*. Caso o valor passado seja inválido, a função retorna o valor *default* informado.

```
head :: a -> [a] -> a
head _ (x:_) = x
head default [ ] = default
```

Tipo Maybe

As soluções apresentadas até o momento para o tratamento de exceções são opostas:

O uso da função **`error`** tem a vantagem de informar ao usuário/programador que a exceção ocorreu (e, possivelmente, a causa dela), mas tem a desvantagem de encerrar a execução do programa;

O uso de um valor fictício tem a vantagem de evitar o encerramento da execução do programa, mas tem a desvantagem de não informar ao usuário/programador que a exceção ocorreu, com o risco de que o programa chegue a um resultado incorreto, caso o valor fictício não seja previsto pelas funções que chamam a função que o retornou.

Tipo Maybe

Seria melhor existir uma solução que permita avisar o usuário do problema e não interromper a exceção do programa.

Por exemplo:

```
fatorial :: Int -> Int
fatorial n = product [1..n]
```

Como definir que a função **`fatorial`** deve retornar algo que indique a ocorrência da exceção, mas ao mesmo tempo não interromper a execução do programa?

Tipo Maybe

Para situações como essa, Haskell oferece o tipo **Maybe**.

Esse tipo **sempre** é usado junto com outro tipo e, **na grande maioria dos casos**, como parâmetro de saída das funções.

Uma função que possui o tipo Maybe como saída pode retornar nada (`Nothing`) ou simplesmente (`Just`) um valor.

```
fatorial :: Int -> Maybe Int
fatorial n =
  | n < 0 = Nothing
  | otherwise = Just (product [1..n])
```

Tipo Maybe

Um valor do tipo **`Maybe x`** não é compatível com um valor do tipo **`x`**.

Por exemplo, a expressão **`6 * (fatorial 5)`** resulta em erro porque não é possível multiplicar **`6`**, que é do tipo **`Int`**, com o resultado de **`fatorial 5`**, que é do tipo **`Maybe Int`**.

Isso pode ser resolvido com a função **`fromJust`**, fornecida pela biblioteca **`Data.Maybe`**.

```
fatorial :: Int -> Maybe Int
fatorial 0 = Just 1
fatorial n
  | n < 0 = Nothing
  | otherwise = Just (n * fromJust (fatorial (n-1)))
```

Tipo Maybe

Da mesma forma, as funções que utilizam uma função que retorna **`Maybe x`** devem estar preparadas para proceder de forma diferente para o caso de obter **`Nothing`** ou **`Just x`**.

```
import Data.Maybe (fromJust)

fatorial :: Int -> Maybe Int
fatorial 0 = Just 1
fatorial n = if n < 0 then Nothing else Just (n * fromJust (fatorial (n-1)))

getFatorial :: Int -> IO (Maybe Int)
getFatorial n = return (fatorial n)
```

Tipo Maybe

```
main = do
  putStr "Digite um inteiro: "
  n <- getLine
  f <- getFatorial (read n)
  if f == Nothing
  then putStr $ n ++ " não é válido para fatorial.\n"
  else putStr $ "O fatorial de " ++ n ++ ": " ++
                show (fromJust f) ++ ".\n"
```

Exercícios

1. Defina uma função **`process :: [Int] -> Int -> Int -> Int`** de forma que **`process lst n m`** toma o **`n`**-ésimo e o **`m`**-ésimo elementos da lista **`lst`** e retorna a soma de todos os elementos entre **`n`** e **`m`**. A função deve retornar zero se quaisquer dos números **`n`** ou **`m`** não forem índices da lista **`lst`** ou se **`n`** for maior que **`m`**.

2. Crie uma função polimórfica que recebe uma lista e um inteiro **`i`** e retorna a substring da posição **`0`** até a posição (**`i-1`**). A função deve lançar uma exceção e encerrar a execução caso **`i`** seja negativo. Se **`i`** for maior que o tamanho da lista, deve-se retornar a lista inteira.

3. Crie uma função polimórfica que recebe uma lista de funções f e uma lista elementos a retorna uma lista do tipo Maybe b, cujos elementos são: [Just f_1 a_1, Just f_2 a_2, …, Just f_n a_n]. Caso as listas de funções e de elementos tenham tamanhos diferentes, o resultado deve ser Nothing para os elementos que não possuam um correspondente na outra lista.