U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Trabalho Prático 1

Haskell

Programação Funcional e em Lógica 2021/2022

T2G04

Filipe Campos up201905609@edu.fe.up.pt
Francisco Cerqueira up201905337@edu.fe.up.pt

Exercício 1 - Fibonacci Functions com Integral

fibRec :: (Integral a) => a -> a

Esta função é responsável por calcular o enésimo número de Fibonacci usando uma implementação recursiva, recorrendo à fórmula de recorrência:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2),\ n > 1$$

Foram utilizados vários casos de teste, tais como:

- Inteiros negativos, onde é obtida a mensagem `"n must be positive"`;
- Casos base (0 e 1), onde os resultados são os esperados;
- Inteiros positivos superiores a 1, onde os resultados são os esperados.

fibLista :: (Integral a) => a → a

Esta função, sendo ela uma versão otimizada da função anterior, utiliza uma lista de resultados parciais tal que lista !! i contém o número de Fibonacci de ordem i. Assim, através de programação dinâmica, calcula todos os elementos até ordem i, retornando-o como resultado para a função principal. A função auxiliar calcula recursivamente a lista de todos os elementos anteriores e acrescenta no final o número cujo resultado é a soma dos dois resultados anteriores.

Foram utilizados os mesmos casos de teste da função anterior, verificando-se um aumento significativo no desempenho desta função.

fibListaInfinita :: (Integral a) => a → a

Esta função segue a mesma lógica da função anterior, com a diferença de que é gerada uma lista "infinita", tirando, assim, proveito do método "Lazy Evaluation" implementado pelo Haskell.

Foram utilizados os mesmos casos de teste das funções anteriores, verificando-se um desempenho semelhante à função anterior e bastante superior à implementação recursiva.

Exercício 2 - Big Numbers

scanner :: String → BigNumber

Esta função lê a string fornecida como argumento e retorna o Big Number correspondente, no caso de ser fornecido uma string que não represente um número decimal será lançado um erro adequado. Para representar um número negativo, a string deverá ser precedida pelo caráter '-'.

Casos de teste:

- Número positivo válido, é retornado o Big Number correspondente como esperado.
    - scanner "52" → BigNumber [2,5] Positive
    - scanner "0" → BigNumber [0] Positive
    - scanner "123456" → BigNumber [6,5,4,3,2,1] Positive
- Número negativo válido, é retornado o BigNumber correspondente como esperado.
    - scanner "-1" → BigNumber [1] Negative
    - scanner "-256" → BigNumber [6,5,2] Negative
- String vazia
    - scanner "" → error "invalid string"
- String inválida
    - scanner "#123" → error "invalid digit"
    - scanner "abc" → error "invalid digit"
    - scanner "-" → error "invalid digit"

output :: BigNumber → String

Esta função reverte a transformação realizada pelo scanner, convertendo um BigNumber na sua representação em String.

Esta função pode ser testada revertendo os números válidos dos testes anteriores para a sua representação em String original.

- Número Positivo
    - output (BigNumber [2,5] Positive) → "52"
    - output (BigNumber [0] Positive) → "0"
    - output (BigNumber [6,5,4,3,2,1] Positive) → "123456"
- Número Negativo
    - output (BigNumber [1] Negative) → "-1"
    - output (BigNumber [6,5,2] Negative) → "-256"

operationWithCarry :: (Int → Int → Int → (Int, Int)) → BigNumberDigits → BigNumberDigits → Int → BigNumberDigits

Função que recebe uma função de tipo (Int → Int → Int → (Int, Int)), dois Big Number Digits, d1 e d2, e um Int que representa o carry, retornando BigNumberDigits. É utilizada uma definição recursiva que percorre, par a par, os dígitos de d1 e d2, começando pelos menos significativos. Para cada par é calculado, utilizando a função fornecida, o dígito resultante para a casa atual e o carry que é fornecido à iteração seguinte.

Esta função é utilizada na implementação das funções somaBN, subBN e mulBN que passam, respectivamente, as funções auxiliares somaOp, subOp, mulBn

- somaOp: (digito, carry) = (a+b+carry % 10, a+b+carry // 10)
- subOp: (digito, carry) = (a-b-carry % 10, if carry + b > a then 1 else 0)
- mulOp: (digito, carry) = ((x*y+carry) % 10, (x*y+carry) // 10)

somaBN :: BigNumber → BigNumber → BigNumber

Esta função realiza a soma entre dois Big Numbers recorrendo à função operationWithCarry, juntamente com a função auxiliar somaOp. Utilizamos pattern matching para delegar o cálculo da soma de números com sinal diferente para a função subBn, deste modo, é apenas necessário calcular a soma de números com o mesmo sinal.

Casos de Teste (Nota: é utilizada a função scanner para tornar os casos de teste mais claros):

- Dois números positivos
    - somaBN (scanner “523”) (scanner “0”) → BigNumber [3,2,5] Positive
    - somaBN (scanner “523”) (scanner “7”) → BigNumber [0,3,5] Positive
    - somaBN (scanner "18") (scanner "8") → BigNumber [6,2] Positive
- Dois números negativos
    - somaBN (scanner “-523”) (scanner “-1”) → BigNumber [4,2,5] Negative
    - somaBN (scanner “-523”) (scanner “-7”) → BigNumber [0,3,5] Negative
    - somaBN (scanner “-18”) (scanner “-8”) → BigNumber [6,2] Negative
- Números com sinais diferentes (calculado por subBN)
    - somaBN (scanner "25") (scanner "-7") → BigNumber [8,1] Positive
    - somaBN (scanner "-25") (scanner "7") → BigNumber [8,1] Negative

subBN :: BigNumber → BigNumber → BigNumber

Função que calcula a subtração de dois big numbers, tal como a somaBN, esta função recorre à operationWithCarry juntamente com uma função auxiliar subOp. Utilizamos pattern matching para simplificar as operações, usufruindo da função somaBN para calcular a subtração de números com sinais diferentes e reduzindo assim as operações num único caso base onde ambos os números são positivos. Também comparamos os dois números para apenas ser necessário realizar a subtração de dois números positivos x,y tal que x > y, usufruindo da função operationWithCarry.

Casos de Teste:

- Dois números positivos
    - subBN (scanner "24") (scanner "17") → BigNumber [7] Positive
    - subBN (scanner “17”) (scanner “24”) → BigNumber [7] Negative
- Dois números negativos
    - subBN (scanner “-24”) (scanner “-17”) → BigNumber [7] Negative
    - subBN (scanner “-17”) (scanner “-24”) → BigNumber [7] Positive
- Números com sinais diferentes (calculado por somaBN)
    - subBN (scanner “523”) (scanner “-7”) → BigNumber [0,3,5] Positive
    - subBN (scanner “-523”) (scanner “7”) → BigNumber [0,3,5] Negative

mulBN :: BigNumber → BigNumber → BigNumber

Função que multiplica dois BigNumbers. Para multiplicar dois números, x e y, primeiro criamos uma lista contendo os resultados da multiplicação de x por cada dígito de y, acompanhado de zeros à direita para representar a ordem de grandeza de cada uma destas multiplicações. Posteriormente todos os sub-resultados presentes na lista são adicionados recorrendo à função somaBN.
Casos de Teste:

- Multiplicação por zero
    - mulBN (scanner “0”) (scanner “123”) → BigNumber [0] Positive
    - mulBN (scanner “123”) (scanner “0”) → BigNumber [0] Positive
- Multiplicação por um
    - mulBN (scanner "123") (scanner "1") → BigNumber [3,2,1] Positive
    - mulBN (scanner "1") (scanner "123") → BigNumber [3,2,1] Positive
- Outros casos
    - mulBN (scanner “64”) (scanner “2”) → BigNumber [8,2,1] Positive
    - mulBN (scanner “64”) (scanner “-2”) → BigNumber [8,2,1] Negative
    - mulBN (scanner “-64”) (scanner “-2”) → BigNumber [8,2,1] Positive
    - mulBN (scanner “128”) (scanner “128”) → BigNumber [4,8,3,6,1] Positive

divBN :: BigNumber → BigNumber → (BigNumber, BigNumber)

Esta função divide dois Big Numbers, retornando o par (quociente, resto).
A função divBN utiliza uma função auxiliar recursiva que sucessivamente compara os dígitos mais significativos do dividendo com o divisor. Enquanto esses dígitos forem menores que o divisor, é recursivamente repetida a comparação incluindo o próximo dígito mais significativo do dividendo.
Após serem encontrados dígitos superiores ao divisor serão testados todos valores do intervalo [divisor*i | i<-[1..9]], de modo a obter o valor a subtrair ao dividendo e o dígito a adicionar ao quociente.
A função termina após terem sido considerados todos os dígitos do dividendo.

Casos de Teste:

- Divisão de zero por um número diferente de zero
    - divBN (scanner "0") (scanner "2") → (BigNumber [0] Positive,BigNumber [0] Positive)
- Divisão de um número por zero
    - divBN (scanner “0”) (scanner “0”) → error “Division by 0”
    - divBN (scanner “123”) (scanner “0”) → error “Division by 0”
- Outros casos
    - divBN (scanner “40”) (scanner “2”) → (BigNumber [0,2] Positive,BigNumber [0] Positive)
    - divBN (scanner “40”) (scanner “25”) → (BigNumber [1] Positive,BigNumber [5,1] Positive)

Exercício 3 - Fibonacci Functions com Big Numbers

fibRecBN :: BigNumber → BigNumber

Esta função, tal como a função fibRec, é responsável por calcular o enésimo número de Fibonacci usando uma implementação recursiva, recorrendo à fórmula de recorrência apresentada previamente, com a diferença de que todas as operações realizadas pela mesma utilizam BigNumbers.

Foram utilizados casos de teste semelhantes à função original, tais como:

- BigNumbers negativos, onde é obtida a mensagem `"BigNumber must be positive";`
- Casos base (BigNumbers 0 e 1), onde os resultados são os esperados;
- BigNumbers positivos superiores a 1, onde os resultados são os esperados.

fibListaBN :: BigNumber → BigNumber

Esta função, tem a mesma estrutura da função fibLista, com a diferença de que todas as operações são realizadas com BigNumbers.

Foram utilizados os mesmos casos de teste da função anterior, verificando-se um aumento significativo no desempenho desta função.

fibListaInfinitaBN :: BigNumber → BigNumber

Tal como na função fibListaInfinita, esta função segue a mesma lógica da função anterior, com a diferença de que é gerada uma lista "infinita", tirando, assim, proveito do método "Lazy Evaluation" implementado pelo Haskell, realizando apenas operações com BigNumbers.

Foram utilizados os mesmos casos de teste das funções anteriores, verificando-se um desempenho semelhante à função anterior e bastante superior à implementação recursiva.

Exercício 4 - Fibonacci Functions Type Comparison

Em Haskell, os valores do tipo Int estão garantidamente contidos no intervalo [-2^29..2^29-1][1] e, em sistemas 64 bit [-2^63..2^63-1].
Por outro lado, tanto o tipo Integer como os Big Numbers utilizam uma implementação baseada em listas, permitindo ter, assim, uma gama de representação "infinita", dependendo apenas da memória alocada ao processo por parte do sistema.

Logo, os valores do tipo Integer e BigNumber permitem uma maior aplicação das funções contidas no ficheiro Fib.hs.

- Int → 44 é o primeiro número cujo resultado é superior a 2^29-1, levando a um overflow. O mesmo problema ocorre para números a partir de 93 (inclusive) num sistema 64 bit.
- Integer → A função fibRec aceita qualquer número Integer. Para as restantes funções, todos os números superiores a (maxBound :: Int) causam overflow ao index usado pela função (!!). Para sistemas de 64 bits este limite é 2^63, e o valor mais alto garantido pela especificação do Haskell é 2^29.
- Big Number → Funciona para qualquer número, visto que não é usada a função (!!) para obter um elemento da lista, já que implementamos uma função auxiliar (indexBN) para a substituir.

1 - https://hackage.haskell.org/package/base-4.16.0.0/docs/Data-Int.html

Exercício 5 - Big Numbers Safe Division

safeDivBN :: BigNumber → BigNumber → Maybe (BigNumber, BigNumber)

Esta função tem um comportamento semelhante ao da função divBN, com a diferença de que o valor de retorno é um Monad tal que, no caso da divisão por zero, seja retornado Nothing e nos restantes casos Just (result).

Casos de Teste:

- Divisão de zero por um número diferente de zero
  - divBN (scanner "0") (scanner "2") → Just (BigNumber [0] Positive,BigNumber [0] Positive)
- Divisão de um número por zero
  - divBN (scanner "0") (scanner "0") → Nothing
  - divBN (scanner "123") (scanner "0") → Nothing
- Outros casos
  - divBN (scanner "40") (scanner "2") → Just (BigNumber [0,2] Positive,BigNumber [0] Positive)
  - divBN (scanner "40") (scanner "25") → Just (BigNumber [1] Positive,BigNumber [5,1] Positive)