PFL - Trabalho 1 - Relatório

Casos de Teste

Funções BigNumbers.hs:

Para as quatro funções principais do módulo *BigNumber* (adição, subtração, multiplicação e divisão), foram considerados os seguintes casos de teste:

- Argumentos pequenos e positivos, onde seria fácil de detetar erros no algoritmo.
- Argumentos pequenos e negativos (exceto na divisão), com o objetivo semelhante ao teste de cima mas desta vez com números negativos.
- Usar 0 num dos argumentos e verificar se o resultado é o esperado em cada caso.
- Argumentos grandes (positivos e negativos quando aplicável) que não poderiam ser usados num tipo *Int*, verificando se o programa é resiliente a estes e verificando o resultado com o auxílio de uma calculadora.
- Por fim, utilizar um argumento positivo e outro negativo, verificando que o programa tem o comportamento correto (exceto na divisão).

Em relação à função *safeDivBN*, foi acrescentado o teste da divisão por 0, verificando que não causa um erro no programa e que retorna `Nothing`.

Funções Fib.hs:

Em todas estas funções, o valor recebido como argumento representa o mesmo resultado, isto é, o indice da posição na sequência de fibonacci. Assim sendo, não temos muitos casos de teste neste contexto. Experimentando com o valor 0 obtemos sempre o caso base definido em todas as funções, que é 0. Os números negativos não são considerados e são tratados como um erro nestas funções, lançando a excepção "negative numbers not accepted". Em termos de valores máximos aceites, isto dependerá de qual o tipo do argumento, o que é discutido na pergunta 4. De resto, foram testados vários números de fibonacci em todas as funções, comparando o resultado ao já conhecido e tabelado.

Explicação das funções

fibRec: Sendo uma função recursiva, colocamos os casos base, neste caso, o valor de fibonnaci de indices 0 e 1. A parte recursiva consiste em somar os 2 elementos anteriores ao valor dado, consecutivamente, dividindo em partes até alcançar os casos base.

fibLista: Nesta função foi particularmente díficil não recorrer a listas infinitas nem a recursão. Acabamos por utilizar uma lista parcial obtida através de um map que utiliza uma função que explicita que cada valor de índice x é equivalente à soma dos de índice x-1 e x-2. Esta função é depois aplicada à lista que vai de 2 até N, ou seja, a lista dos indíces, excluindo os 2 primeiros elementos que ja são colocados com o operador `:`.

fibListaInfinita: Para esta função encontrámos rapidamente uma forma simples e intuitiva de satisfazer o que é pedido.Através da função iterate iteramos por um tuplo (0,1) inicial, i.e, os 2 primeiros elementos da sucessão com a função lambda (x,y) -> (y,x+y) o que nos devolve tuplos infinitos nos quais o primeiro elemento dum tuplo corresponde ao sucessor do primeiro elemento do tuplo anterior.Sendo assim, com map fst sobre este conjunto de tuplos chegamos a uma lista infinita da sucessão de fib.

fibRecBN: A função fibRec foi facilmente adaptada aos BigNumbers, bastou alterar os casos base para esse formato e substituir as operações aritméticas na chamada recursiva pelas que se encontravam na biblioteca de BigNumbers.

fibListaBN: Provavelmente a mais desafiante dado o facto de nos ser pouco intuitivo utilizar listas parciais, juntamente com a dificuldade da nova estrutura. Tivemos de definir 2 funções auxiliares, a primeiro foi a *nthBN*, cujo nome é auto explicativo, tendo sido baseada na definição prelude de *"!!"*. Necessitamos de outra função auxiliar que substituísse o gerador [2..n] para listas parciais, a *rangeBN*, que nos permite fazer uma lista de *BigNumbers* que vai do primeiro argumento até ao segundo. Após isso, conseguimos adaptar a nossa *fibLista* normal ao novo tipo de dados.

fibListaInfinitaBN: Esta função também foi facilmente adaptada a partir da que definimos inicialmente, apenas substituindo as operações aritméticas pelas da biblioteca e os dois primeiros elementos iniciais no tuplo pelo seu equivalente em *BigNumbers*.

output: Converte um *BigNumber* para uma *String*, convertendo cada dígito para um caráter (usando *intToDigit*) e pondo '-' caso o número seja engativo.

scanner: Converte uma *String* para um *BigNumber*, convertendo cada caráter para um dígito (usando *digitToInt*). Cria um negativo se houve um '-' no início da string.

somaBN: Soma dois *BigNumber*'s. Se os sinais destes forem iguais, soma os valores absolutos e mantém-no. Caso contrário, subtrai o valor absoluto menor ao maior e mantém o sinal do maior.

subBN: Subtrai dois *BigNumber*'s. Se os sinais destes forem diferentes, soma os seus valores absolutos e mantém o sinal do primeiro. Caso contrário, subtrai o valor absoluto menor ao maior e mantém o sinal do maior.

mulBN: Multiplica dois *BigNumber*'s. Multiplica os seus valores absolutos e, caso os sinais dos números sejam iguais, retorna um nº positivo. Caso contrário, retorna um negativo.

divBN: Divide dois *BigNumber*'s, retornando o quociente e o resto, sendo que aceita apenas números positivos.

safeDivBN: Utiliza o tipo *Maybe* para dividir segura de dois *BigNumber*'s. Chama a função *divBN* para todos os casos exceto quando o denominador é 0. Neste caso, retorna *Nothing*.

Estratégias de BigNumber

SomaBN

Antes de tudo, é feita uma análise aos argumentos dados que verifica se têm o mesmo sinal. Se sim, esse sinal é mantido e é chamada uma função que calcula a soma dos seus valores absolutos. Caso contrário, é mantido o sinal do número maior (em módulo) e chamada uma função que subtrai o valor absolute menor ao maior. Se eles forem simétricos, é simplesmente retornado 0.

A função que calcula a soma de dois valores absolutos (lista de inteiros) implementa o algoritmo clássico da resolução manual de adições. Ou seja, os algarismos são somados um a um usando `zipWith` (normalizando-os antes para terem o mesmo tamanho), calculando os *carries* em cada uma das somas. Depois, é utilizado um `foldl` para atualizar o valor das somas com os *carries* respetivos. É necessário um especial cuidado para garantir que os *carries* seguintes são atualizados após somar o atual.

A função que calcula a subtração de dois valores absolutos segue uma estratégia semelhante à da soma, assumindo que o primeiro argumento nunca é menor que o segundo. Tal como na soma, é usado `zipWith` para subtrair os dígitos um a um e guardar o seu *carry*. Após isto, o `foldl` também usado mas, desta vez, para atualizar os valores das subtrações, mantendo o mesmo cuidado da soma.

SubBN

Esta função utiliza as mesmas funções auxiliares que *somaBN*, diferenciando-se apenas na análise inicial dos argumentos. Assim sendo, se os sinais destes forem diferentes, é mantido o sinal do primeiros e os seus valores absolutos são somados. Caso contrário, é mantido o sinal do maior número (em módulo) e subtrai-se o valor absoluto do menor. Se eles forem iguais, é simplesmente retornado 0.

MulBN

Antes de tudo, esta função decide o sinal do resultado, sendo positivo caso os sinais dos argumentos sejam iguais e negativo caso contrário. Depois, usa uma função auxiliar para multiplicar os valores absolutos dos dois números.

A estratégia desta função auxiliar passa pelos seguintes passos principais:

- Multiplicar cada dígito do segundo argumento ao primeiro, adicionando 0's no final do resultado, de acordo com a posição do dígito no número inicial. Para tal, é usado um `map` para multiplicar cada dígito e calcular o seu *carry*. Depois, é usada a mesma técnica que na soma para somar os *carries*.
- Guardar todos os resultados anteriores numa lista.
- Somar todos os números que foram guardados na lista. Para tal, usou-se um `foldl` que começa em 0 e reutiliza a função de soma de valores absolutos para adicionar todos os valores na lista.
- O resultado final será retornado pelo `foldl` anterior.

DivBN

Esta função aceita apenas números positivos, pelo que tem apenas que dividir os valores absolutos dos dois números que lhe são passados.
Isto é feito com a seguinte estratégia:

- Se o denominador for maior que o numerador, retornar quociente 0 e resto igual ao numerador.
- Caso contrário, separar o numerador para obter uma parte mais significativa com o mesmo nº de dígitos do denominador (P1). A outra parte será P2.
- Passar as duas partes obtidas para uma função recursiva que vai calculando o quociente em cada iteração, com a seguinte lógica:
    - Calcular o quociente e o resto da divisão entre P1 e o denominador, utilizando uma função que utiliza a subtração. [1]
    - Se P2 for uma lista vazia, retornar o quociente até agora guardado concatenado com este último e o resto obtido nesta operação.
    - Caso contrário, concatenar o quociente obtido ao que já se encontra guardado, usar o resto obtido concatenado com o primeiro dígito de P2 como o novo P1, a `tail` de P2 como novo P2 e o mesmo denominador, voltando a chamar a função.
- Após este procedimento, obtemos o quociente e o resto pretendidos.

[1] Não se utiliza esta função para a divisão total porque escala de forma **muito ineficiente**.

SafeDivBN

A diferença desta função para *divBN* é o facto de retornar um *monad* do tipo `Maybe`. Assim sendo, retorna `Nothing` no caso de o denominador passado ser 0 e simplesmente chama *divBN* nos casos restantes, retornando `Just` com o resultado dessa chamada.

Resposta à alínea 4

Na resolução da alínea 1, podem ser usados os valores **Int** ou **Integer**, visto que as funções aceitam um argumento da classe **Integral**.

Em relação às funções chamadas com o tipo (Int -> Int), estas aceitam argumentos entre 0 e o máximo definido pelo interpretador de Haskell (este sendo 9223372036854775807 no nosso caso). Embora este número já seja bastante grande, não é possível calcular números de Fibonacci maiores.

Em relação ao tipo (Integer -> Integer), este já aceita qualquer número positivo, desde que a memória da máquina o permita. No entanto, isto só é verdade para a função *fibRec*, visto que as outras duas utilizam o operador **!!** que aceita um **Int**. Isto quer dizer que, mesmo que se use **Integer**, as funções só vão aceitar até o máximo de **Int**.

Por sua vez, as funções da alínea 3 têm o tipo (BigNumber -> BigNumber), o que quer dizer que aceitam qualquer número positivo, em qualquer uma das funções. Esta é a melhor alternativa para calcular números de Fibonacci maiores do que o máximo de **Int**, visto que *fibRec* é uma função pouco eficiente.