Representando vetores matemáticos em Haskell com Tipos de Dados Algébricos

Vetores matemáticos são uma estrutura muito útil na computação, podem ser usados em simulações fisícas, computação gráfica, jogos, entre outros. A maioria das aplicações utiliza vetores de 2 ou 3 dimensões.

Sendo assim, este tutorial tem como objetivo respresentar vetores de 2 e 3 dimensões em Haskell utilizado Tipos de Dados Algébricos. Além disso vamos implementar as seguintes operações de vetores:

- Soma
- Subtração
- Multiplicação elemento a elemento (vamos chamar apenas de multiplicação durante este tutorial, importante não confundir com Produto Vetorial)
- Divisão
- Produto Escalar
- Magnitude

Versão Inicial

Podemos começar uma versão inicial do Vetor assim:

```haskell
data Vector2 = Vector2 Double Double
    deriving (Show)
```

Ela é um TDA (Tipo de Dado Algébrico) produto, que armazena 2 valores diferentes de Double ao mesmo tempo, com ele podemos representar as duas dimensões do vetor. Além disso, derivamos a Classe de Tipos *Show*, para que o compilador gere automaticamente a função para converter a estrutura em texto.

Podemos construir a o TDA utilizando o constructor Vector2(Possui o mesmo nome do Vector2, mas é uma função):

```
ghci> Vector2 1 2
Vector2 1.0 2.0
ghci> Vector2 2 3
Vector2 2.0 3.0
ghci> Vector2 5 6
Vector2 5.0 6.0
```

Um problema deste construtor é que ele não permite acessar cada um dos elementos por nome, apenas por pattern matching, mas podemos resolver isso definindo o tipo com a sintaxe de Record Type. Essa sintaxe cria automaticamente funções com o nome do elemento que recebem o vetor e retornam o valor desse elemento.

```haskell
data Vector2 = Vector2 { x :: Double,
                         y :: Double
                       }
    deriving (Show)
```

Dessa forma podemos acessar as dimensões do vetor individualmente.

```
ghci> v = Vector2 5 12
ghci> x v
5.0
ghci> y v
12.0
```

Operações

Agora precisamos criar operações para dar utilidade a este vetor.

As operações de Soma, Subtração, Multiplicação e Divisão todas funcionam da mesma forma, aplicando a mesma operação de tipos numéricos elemento a elemento no vetor, então podemos criar uma função auxililar que abstrai esse processo, chamaremos de **memberToMemberOperation**, e vamos usá-la para criar as 4 operações supracitadas:

```haskell
memberToMemberOperation :: (Double -> Double -> Double) -> Vector2 ->
Vector2 -> Vector2
memberToMemberOperation operation (Vector2 x1 y1) (Vector2 x2 y2) =
    Vector2 (operation x1 x2) (operation y1 y2)

sumVec2 :: Vector2 -> Vector2 -> Vector2
sumVec2 v1 v2 = memberToMemberOperation (+) v1 v2

subVec2 :: Vector2 -> Vector2 -> Vector2
subVec2 v1 v2 = memberToMemberOperation (-) v1 v2

multVec2 :: Vector2 -> Vector2 -> Vector2
multVec2 v1 v2 = memberToMemberOperation (*) v1 v2

divVec2 :: Vector2 -> Vector2 -> Vector2
divVec2 v1 v2 = memberToMemberOperation (/) v1 v2
```

```
ghci> v1 = Vector2 12 21
ghci> v2 = Vector2 2 3
ghci> sumVec2 v1 v2
Vector2 {x = 14.0, y = 24.0}
ghci> subVec2 v1 v2
Vector2 {x = 10.0, y = 18.0}
ghci> multVec2 v1 v2
```

```
Vector2 {x = 24.0, y = 63.0}
ghci> divVec2 v1 v2
Vector2 {x = 6.0, y = 7.0}
```

Por fim, definimos individualmente as operações restantes:

```haskell
dotProduct :: Vector2 -> Vector2 -> Double
dotProduct (Vector2 x1 y1) (Vector2 x2 y2) = x1 * x2 + y1 * y2

magnitude :: Vector2 -> Double
magnitude v1 = sqrt $ dotProduct v1 v1
```

Suporte a tipos genéricos

Um pequena melhoria a esse vetor, é suportar tipos diferentes para os valores internos, podemos fazer isso utilizando tipos paramétricos em vez de fixar o tipo do vetor como Double.

Primeiro alteramos o tipo Vector2 para receber um parametro genérico a para os tipos de x e y

```haskell
data Vector2 a = Vector2  { x :: a,
                            y :: a
                          }

    deriving (Show)
```

Depois precisamos modificar todas as funções que recebem um **Vector2** para receber um **Vector2 a**, e todas que retornam um **Double** para retornar um **a**.

Além disso, é necessário criar restrições para os tipos das funções, isso é feito usando os constraints como primeiro parametro da função. A maioria das funções pode funcionar para qualquer Vector2 cujo tipo interno seja da classe de tipos **Num**, entretanto *divVec2* e *magnitude* dependem de funcões exclusivas das classe de tipos **Fractional** e **Floating** respectivamente, sendo assim precisamos restringi-las a essas classes de tipos. Para não perder o operador de divisão, criamos um operador especifico para tipos inteiros, o *integralDivVec2*.

```haskell
memberToMemberOperation :: Num a => (a -> a -> a) -> Vector2 a -> Vector2 a
-> Vector2 a
memberToMemberOperation operation (Vector2 x1 y1) (Vector2 x2 y2) =
    Vector2 (operation x1 x2) (operation y1 y2)

sumVec2 :: Num a => Vector2 a -> Vector2 a -> Vector2 a
sumVec2 v1 v2 = memberToMemberOperation (+) v1 v2

subVec2 :: Num a => Vector2 a -> Vector2 a -> Vector2 a
subVec2 v1 v2 = memberToMemberOperation (-) v1 v2

multVec2 :: Num a => Vector2 a -> Vector2 a -> Vector2 a
```

```
multVec2 v1 v2 = memberToMemberOperation (*) v1 v2

divVec2 :: Fractional a => Vector2 a -> Vector2 a -> Vector2 a
divVec2 v1 v2 = memberToMemberOperation (/) v1 v2

integralDivVec2 :: Integral a => Vector2 a -> Vector2 a -> Vector2 a
integralDivVec2 v1 v2 = memberToMemberOperation (div) v1 v2

dotProduct :: Num a => Vector2 a -> Vector2 a -> a
dotProduct (Vector2 x1 y1) (Vector2 x2 y2) = x1 * x2 + y1 * y2

magnitude :: Fractional a => Vector2 a -> a
magnitude v1 = sqrt $ dotProduct v1 v1
```

```
ghci> v1 = Vector2 (1::Int) (2::Int)
ghci> :t v1
v1 :: Vector2 Int
ghci> v2 = Vector2 (10::Int) (20::Int)
ghci> sumVec2 v1 v2
Vector2 {x = 11, y = 22}
ghci> subVec2 v1 v2
Vector2 {x = -9, y = -18}
ghci> multVec2 v1 v2
Vector2 {x = 10, y = 40}
integralDivVec2  interact
ghci> integralDivVec2 v2 v1
Vector2 {x = 10, y = 10}
ghci> :t x v1
x v1 :: Int
ghci> dotProduct v1 v2
50
```

```
ghci> v1 = Vector2 1.0 2.0
ghci> :t v1
v1 :: Fractional a => Vector2 a
ghci> v2 = Vector2 33.0 75.6
ghci> sumVec2 v1 v2
Vector2 {x = 34.0, y = 77.6}
ghci> sub
subVec2   subtract
ghci> subVec2 v1 v2
Vector2 {x = -32.0, y = -73.6}
ghci> multVec2 v1 v2
Vector2 {x = 33.0, y = 151.2}
ghci> div
div      divMod   divVec2
ghci> divVec2 v1 v2
Vector2 {x = 3.0303030303030304e-2, y = 2.6455026455026457e-2}
ghci> magnitude v1
```

```
2.23606797749979
ghci> magnitude v2
82.48854465924343
ghci> dotProduct v1 v2
184.2
```

Suporte a 3 dimensões

Por fim, precisamos fazer com que o TDA functione tanto para 2 quanto para 3 dimensões, para isso podemos transformar nosso TDA em um Tipo Soma, de forma a ter 2 construtores, um para construir um Vetor de 2 dimensões, e um para construir um vetor de 3 dimensões. Como nosso TDA não suporta apenas 2 dimensões mais, podemos renomeá-lo para apenas **Vector**.

```
data Vector a = Vector2 { x :: a,
                          y :: a
                        } |
                Vector3 { x :: a,
                          y :: a,
                          z :: a
                        }
    deriving (Show)
```

Feito isso, ainda precisamos modificar algumas funções para que se adaptem a trabalhar com vetores de 3 dimensões, o que pode ser feito com pattern matching. Com o tratamento do caso de 3 dimensões na função *memberToMemberOperation*, ganhamos automaticamente uma implementação dos operadores de soma, subtração, multiplicação e divisão sem nenhuma modificação (só trocar o nome dessas funções para que reflitam a natureza do novo vetor que não é exclusivamente bidimensional).

```
memberToMemberOperation :: Num a => (a -> a -> a) -> Vector a -> Vector a -
> Vector a
memberToMemberOperation operation (Vector2 x1 y1) (Vector2 x2 y2) =
    Vector2 (operation x1 x2) (operation y1 y2)
memberToMemberOperation operation (Vector3 x1 y1 z1) (Vector3 x2 y2 z2) =
    Vector3 (operation x1 x2) (operation y1 y2) (operation z1 z2)

sumVec :: Num a => Vector a -> Vector a -> Vector a
sumVec v1 v2 = memberToMemberOperation (+) v1 v2

subVec :: Num a => Vector a -> Vector a -> Vector a
subVec v1 v2 = memberToMemberOperation (-) v1 v2

multVec :: Num a => Vector a -> Vector a -> Vector a
multVec v1 v2 = memberToMemberOperation (*) v1 v2

divVec :: Fractional a => Vector a -> Vector a -> Vector a
divVec v1 v2 = memberToMemberOperation (/) v1 v2
```

```
integralDivVec :: Integral a => Vector a -> Vector a -> Vector a
integralDivVec v1 v2 = memberToMemberOperation (div) v1 v2
```

O mesmo ocorre com a função *dotProduct*, fazendo com que ela suporte vetores tridimensionais, ganhamos uma implementação da função *magnitude* para 3 dimensões.

```
dotProduct :: Num a => Vector a -> Vector a -> a
dotProduct (Vector2 x1 y1) (Vector2 x2 y2) = x1 * x2 + y1 * y2
dotProduct (Vector3 x1 y1 z1) (Vector3 x2 y2 z2) = x1 * x2 + y1 * y2 + z1 *
z2

magnitude :: Floating a => Vector a -> a
magnitude v1 = sqrt $ dotProduct v1 v1
```

Com isso, temos uma biblioteca para vetores de 2 e 3 dimensões em haskell utilizando tipos de dados algébricos.

```
ghci> v1 = Vector3 1 2 3
ghci> v2 = Vector3 4 5 6
ghci> sumVec v1 v2
Vector3 {x = 5, y = 7, z = 9}
ghci> subVec v1 v2
Vector3 {x = -3, y = -3, z = -3}
ghci> multVec v1 v2
Vector3 {x = 4, y = 10, z = 18}
ghci> divVec v1 v2
Vector3 {x = 0.25, y = 0.4, z = 0.5}
ghci> magnitude v1
3.7416573867739413
ghci> magnitude v2
8.774964387392123
ghci> dotProduct v1 v2
32
ghci> v3 = Vector2 5 6
ghci> v4 = Vector2 66 77
ghci> sumVec v3 v4
Vector2 {x = 71, y = 83}
ghci> integralDivVec v3 v4
Vector2 {x = 0, y = 0}
ghci> integralDivVec v4 v3
Vector2 {x = 13, y = 12}
ghci>
```

Código Fonte Completo

module Lib (Vector (..), sumVec, subVec, multVec, integralDivVec, dotProduct, magnitude) where

data Vector a = Vector2 { x :: a, y :: a } | Vector3 { x :: a, y :: a, z :: a } deriving (Show)

```
memberToMemberOperation :: Num a => (a -> a -> a) -> Vector a -> Vector a -
> Vector a
memberToMemberOperation operation (Vector2 x1 y1) (Vector2 x2 y2) =
    Vector2 (operation x1 x2) (operation y1 y2)
memberToMemberOperation operation (Vector3 x1 y1 z1) (Vector3 x2 y2 z2) =
    Vector3 (operation x1 x2) (operation y1 y2) (operation z1 z2)

sumVec :: Num a => Vector a -> Vector a -> Vector a
sumVec v1 v2 = memberToMemberOperation (+) v1 v2

subVec :: Num a => Vector a -> Vector a -> Vector a
subVec v1 v2 = memberToMemberOperation (-) v1 v2

multVec :: Num a => Vector a -> Vector a -> Vector a
multVec v1 v2 = memberToMemberOperation (*) v1 v2

divVec :: Fractional a => Vector a -> Vector a -> Vector a
divVec v1 v2 = memberToMemberOperation (/) v1 v2

integralDivVec :: Integral a => Vector a -> Vector a -> Vector a
integralDivVec v1 v2 = memberToMemberOperation (div) v1 v2

dotProduct :: Num a => Vector a -> Vector a -> a
dotProduct (Vector2 x1 y1) (Vector2 x2 y2) = x1 * x2 + y1 * y2
dotProduct (Vector3 x1 y1 z1) (Vector3 x2 y2 z2) = x1 * x2 + y1 * y2 + z1 *
z2

magnitude :: Floating a => Vector a -> a
magnitude v1 = sqrt $ dotProduct v1 v1
```