De listas indexadas a redes neurais: segurança no nível dos tipos em Haskell

Matrizes são estruturas de dados muito utilizadas em computação científica. No entanto, lidar com essas estruturas pode ser complicado, pois sempre temos que ter em mente as dimensões de cada estrutura na hora de realizar operações com elas.

Para facilitar a criação de programas que lidam com matrizes, podemos utilizar as capacidades de programação no nível de tipos da linguagem Haskell. Neste texto, vamos explorar como podemos criar estruturas de dados com garantias no nível dos tipos e como podemos aumentar ainda mais essa segurança para lidar com dimensões fornecidas em tempo de execução.

Lista indexável

Antes de falarmos propriamente sobre matrizes, vamos falar sobre listas indexáveis. Uma lista indexável é uma lista no qual podemos acessar qualquer um de seus elementos através de um número inteiro, que representa a posição do elemento na lista. Em Haskell, já possuímos uma estrutura de dados linear indexável:

```haskell
data [a] = [] | a : [a]
```

cuja operação de indexação é feita através da função `!!`:

```haskell
(!!) :: [a] -> Int -> a
(x:_) !! 0 = x
(_:xs) !! n = xs !! (n-1)
```

Aparentemente, tudo funciona como esperado. No entanto, essa definição de lista não é nada segura, pois podemos tentar acessar um índice que não existe na lista, o que resulta em um erro em tempo de execução. O mesmo ocorre para outras funções da lista, como `head` e `tail` que podem falhar se a lista estiver vazia.

Usando programação no nível dos tipos, podemos tornar a lista indexável mais segura. Para isso, vamos definir um novo tipo de lista:

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE StandaloneKindSignatures #-}

import Data.Kind

type ArrayN :: Nat -> Type -> Type
data ArrayN n a where
  Nil :: ArrayN Z a
  (:>) :: a -> ArrayN n a -> ArrayN (S n) a
```

```
infixr 5 :>
```

Essa implementação depende ainda de uma definição de números naturais no nível dos tipos, que é usada para representar, em tempo de compilação, o tamanho da lista. Para isso, vamos definir o tipo `Nat` usando números de Peano:

```
data Nat = Z | S Nat

-- Z <-> 0
-- S Z <-> 1
-- S (S Z) <-> 2
-- ...
```

Agora, podemos definir funções seguras para acessar a cabeça e a cauda da lista por exemplo:

```
lhead :: ArrayN (S n) a -> a
lhead (x :> _) = x

ltail :: ArrayN (S n) a -> ArrayN n a
ltail (_ :> xs) = xs
```

Nessas definições, a própria assinatura da função delimita quais listas podem ser passadas como argumento. As duas funções só aceitam listas com pelos menos um elemento, o que garante, em tempo de compilação, que nunca tentaremos acessar a cabeça ou a cauda de uma lista vazia.

Como essa leve introdução ao mundo da programação no nível de tipos feita, podemos partir para implementação de tipos mais complexos, como matrizes.

Matrizes

Uma matriz é uma estrutura de dados bidimensional, ou seja, toda matriz possui uma quantidade de linhas e uma quantidade de colunas, ambas representadas por números inteiros. Assim, podemos utilizar a seguinte definição para representar uma matriz em Haskell:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE StandaloneKindSignatures #-}

import Data.Kind

type Matrix :: Nat -> Nat -> Type -> Type
data Matrix n m a where
  M :: [[a]] -> Matrix n m a
```

Perceba que, assim como no caso da lista, a informação sobre as dimensões da matriz existe apenas no mundo dos tipos. Isso significa que, em tempo de compilação, o compilador pode decidir se uma operação com matrizes é válida ou não.

Vamos criar funções para somar, subtrair, escalar e transpor matrizes:

```haskell
addM :: (Num a) => Matrix r c a -> Matrix r c a -> Matrix r c a
addM (M xs) (M ys) = M $ zipWith (zipWith (+)) xs ys

(#+#) :: (Num a) => Matrix r c a -> Matrix r c a -> Matrix r c a
(#+#) = addM

scaleM :: (Num a) => a -> Matrix r c a -> Matrix r c a
scaleM x (M xs) = M $ map (map (x *)) xs

(*#) :: (Num a) => a -> Matrix r c a -> Matrix r c a
(*#) = scaleM

subM :: (Num a) => Matrix r c a -> Matrix r c a -> Matrix r c a
subM m n = addM m (scaleM (-1) n)

(#-#) :: (Num a) => Matrix r c a -> Matrix r c a -> Matrix r c a
(#-#) = subM

transposeM :: Matrix r c a -> Matrix c r a
transposeM (M xs) = M $ foldr (zipWith (:)) (repeat []) xs

(\^/) :: Matrix r c a -> Matrix c r a
(\^/) = transposeM

mulM :: (Num a) => Matrix r c a -> Matrix c c' a -> Matrix r c' a
mulM (M xs) ys = M $ map (\row -> map (sum . zipWith (*) row) ys') xs
  where
    (M ys') = transposeM ys

(#*#) :: (Num a) => Matrix r c a -> Matrix c c' a -> Matrix r c' a
(#*#) = mulM
```

Em todas as funções, as operações são válidas no nível dos tipos, ou seja, é impossível, por exemplo, somarmos matrizes de dimensões diferentes, pois o compilador não permitirá a compilação do programa caso tentemos usar a função `addM` com matrizes de dimensões diferentes.

Para entender ainda melhor isso, vamos pegar o exemplo da função de multiplicação de matrizes. A multiplicação de matrizes só é válida se o número de colunas da primeira matriz for igual ao número de linhas da segunda matriz. Assim, o compilador pode garantir que a multiplicação de matrizes só será feita se as dimensões das matrizes forem compatíveis. Isso é codificado na assinatura da função `mulM`:

```haskell
mulM :: (Num a) => Matrix r c a -> Matrix c c' a -> Matrix r c' a
-- r -> número de linhas da primeira matriz
```

```
-- c -> número de colunas da primeira matriz
-- c' -> número de colunas da segunda matriz
```

Perceba que o número de linhas da primeira matriz é representado pelo mesmo tipo do número de colunas da primeira matriz `c`. O mesmo vale para matriz resultante.

Aplicações práticas

Atualmente, um dos usos mais comuns de matrizes é nas computações envolvendo redes neurais. Redes neurais são modelos computacionais inspirados no funcionamento do cérebro humano e são utilizados para resolver diversos tipos de problemas, como reconhecimento de imagens, tradução automática, entre outros.

Em redes neurais, as matrizes são utilizadas para representar os pesos das conexões entre os neurônios de camadas diferentes, além de representar os valores de ativação de cada neurônio em determinada camada. Assim, podemos usar a implementação de matrizes que fizemos para criar uma estrutura de dados que represente uma rede neural e garantir, em tempo de compilação, que todas as operações feitas com essa rede são válidas.

As camadas da rede neural

Antes de definirmos a rede neural, vamos definir como funcionam as camadas da rede neural. Cada camada da rede possui `n` neurônios, cada um com um bias associado. Além disso, cada neurônio possui `m` conexões com os neurônios da próxima camada. Assim, podemos representar os pesos das conexões entre os neurônios de uma camada e os neurônios da próxima camada como uma matriz de dimensões `n x m`, e os valores de ativação dos neurônios de uma camada como um vetor de dimensão `n`. Já temos essas duas estruturas definidas!

Agora, vamos considerar como podemos representar nossa rede no nosso programa. Como cada camada depende do número de conexões com a próxima camada, podemos definir um tipo de dados que represente isso:

```
type NeuralNetwork :: Nat -> [Nat] -> Nat -> Type
data NeuralNetwork inputs hidden outputs where
  Output :: Matrix inputs outputs Double -> ArrayN inputs Double -> NeuralNetwork
inputs '[] outputs
  Hidden ::
    Matrix inputs hidden Double ->
    ArrayN hidden Double ->
    NeuralNetwork inputs hidden' outputs ->
    NeuralNetwork inputs (hidden : hidden') outputs
```

Para simplificar esse código, vamos criar uma estrutura de dados que armazene os pesos e os valores de ativação de uma camada:

```
type Layer :: Nat -> Nat -> Type
data Layer inputs outputs where
```

```
  L :: Matrix outputs inputs Double -> ArrayN outputs Double -> Layer inputs
outputs
```

E para codificar ainda melhor a lógica de sequenciamento das camadas, vamos criar um operador que adiciona uma camada à rede neural:

```
type NeuralNetwork :: Nat -> [Nat] -> Nat -> Type
data NeuralNetwork inputs hidden outputs where
  O :: Layer inputs outputs -> NeuralNetwork inputs '[] outputs
  H ::
    Layer inputs hidden ->
    NeuralNetwork hidden hidden' outputs ->
    NeuralNetwork inputs (hidden : hidden') outputs

(#>) :: Layer inputs hidden -> NeuralNetwork hidden hidden' outputs ->
NeuralNetwork inputs (hidden : hidden') outputs
layer #> network = H layer network
```

Antes de continuarmos, vamos primeiro entender o porque essa estrutura representa uma rede neural segura no nível dos tipos. Vamos supor um exemplo de uma rede neural com 3 neurônios de entrada, 2 camadas escondidas com 4 e 5 neurônios, respectivamente, e 2 neurônios de saída. Podemos representar essa rede neural da seguinte forma:

```
l1 :: Layer 3 4
l2 :: Layer 4 5
l3 :: Layer 5 2

nn :: NeuralNetwork 3 '[4, 5] 2
```

A primeira camada representa as conexões dos 3 neurônios de input e dos 4 neurônios da primeira camada escondida. A segunda camada, por sua vez, representa as conexões dos 4 neurônios da primeira camada escondida e dos 5 neurônios da segunda camada escondida. Por fim, a terceira camada representa as conexões dos 5 neurônios da segunda camada escondida e dos 2 neurônios de saída.

Como os tamanhos de cada matriz de pesos e de cada vetor de ativação são definidos no nível dos tipos, o compilador garante que todas as operações feitas com essa rede neural são válidas. Por exemplo, se tentarmos multiplicar a matriz de pesos da primeira camada por um vetor (lembre que vetores são matrizes com apenas uma linha) de ativação de tamanho diferente, o compilador não permitirá a compilação do programa.

Uma das capacidades do Haskell que nos permite realizar esse tipo de operação segura no nível dos tipos é a extensão `TypeFamilies`. Estamos utilizando essa extensão na definição da nossa rede neural quando estamos adicionando um valor a lista de quantidades de neurônios das camadas escondidas. Isso é feito através da família de tipos `::`:

```haskell
type family (:) (x :: a) (xs :: [a]) :: [a] where
  x : xs = x ': xs
```

Se essa definição te lembra de como o operador funciona no nível dos termos, é porque no nível dos tipos acontece exatamente a mesma coisa. Type families não nada mais que funções no nível dos tipos, que são avaliadas em tempo de compilação.

Entendendo a propagação de valores em uma rede neural

Agora que conseguimos representar uma rede neural, precisamos entender como tudo se encaixa. A propagação de valores em uma rede neural é feita através de uma série de operações matriciais e do uso de funções de ativação. Para fins de simplicidade, vamos considerar que a função de ativação de cada neurônio é a função tangente hiperbólica:

```haskell
tanh :: Double -> Double
tanh x = (exp x - exp (-x)) / (exp x + exp (-x))
```

A propagação de valores é feita da seguinte forma:

1. O vetor de entrada é multiplicado pela matriz de pesos da primeira camada e o bias é adicionado ao resultado;
2. O resultado é passado pela função de ativação tangente hiperbólica;
3. O processo é repetido para todas as camadas da rede neural, até que o vetor de saída seja obtido.

Vamos implementar a função de propagação de valores:

```haskell
instance Functor (Matrix r c) where
  fmap f (M xs) = M $ map (map f) xs

propagate :: Layer inputs outputs -> Matrix (S Z) inputs Double -> Matrix (S Z)
outputs Double
propagate (L weights biases) inputs = tanh <$> transposeM z
  where
    z = weights #*# transposeM inputs #+# transposeM biases
```

Lembrando que nessa implementação, estamos utilizando uma definição de `Nat` baseada em números de Peano, por isso estamos utilizando `S Z` para representar o número 1. (Se estivéssemos utilizando o módulo `GHC.TypeLits`, poderíamos utilizar `1` diretamente, mas perderíamos as propriedades indutivas dos números de Peano.)

Para simplificar nossa implementação, invés de usarmos listas `ArrayN` que criamos anteriormente, estamos utilizando matrizes linhas, que, para todos os fins, são equivalentes a vetores. Assim, a função `propagate` recebe um vetor de entrada e retorna um vetor de saída, o qual será utilizado como entrada para a próxima camada.

Processo de retropropagação

Depois que fazemos a propagação de valores na rede neural, precisamos ajustar os pesos das conexões entre os neurônios para que a rede neural realmente aprenda a realizar a tarefa que queremos. Esse processo é chamado de retropropagação e é feito através do algoritmo de gradiente descendente. Como a ideia desse texto é mostrar como podemos garantir a segurança no nível dos tipos, não vou explicar o algoritmo de gradiente descendente:

```
run :: NeuralNetwork inputs hidden outputs -> Matrix (S Z) inputs Double -> Matrix
(S Z) outputs Double
run (O layer) inputs = tanh <$> propagate layer inputs
run (H layer network) inputs = run network (propagate layer inputs)

newtype NeuralNetworkConfig = NeuralNetworkConfig
  { learningRate :: Double
  }

weights :: Layer inputs outputs -> Matrix outputs inputs Double
weights (L w _) = w

biases :: Layer inputs outputs -> Matrix (S Z) outputs Double
biases (L _ b) = b

train :: NeuralNetworkConfig -> NeuralNetwork inputs hidden outputs -> Matrix (S
Z) inputs Double -> Matrix (S Z) outputs Double -> NeuralNetwork inputs hidden
outputs
train config (O layer) inputs targets = O layer'
  where
    y = propagate layer inputs
    o = tanh <$> y
    e = (o #-# targets) !! 0
    d = e #*# (1 - y ^ 2)
    deltaWeights = scaleM (learningRate config) (d #*# transposeM inputs)
    deltaBiases = scaleM (learningRate config) d
    weights' = weights layer #-# deltaWeights
    biases' = biases layer #-# deltaBiases
    layer' = L weights' biases'
train config (H layer network) inputs targets = H layer' (train config network
(propagate layer inputs) targets)
  where
    y = propagate layer inputs
    o = tanh <$> y
    e = (o #-# targets) !! 0
    d = e #*# (1 - y ^ 2)
    deltaWeights = scaleM (learningRate config) (d #*# transposeM inputs)
    deltaBiases = scaleM (learningRate config) d
    weights' = weights layer #-# deltaWeights
    biases' = biases layer #-# deltaBiases
    layer' = L weights' biases'
```

Cruzando o mundo dos tipos para o mundo dos valores

Temos uma implementação completa de uma rede neural, com seguranças garantidas em nível dos tipos. É impossível criarmos uma rede neural com dimensões inválidas, pois o compilador não permitirá a compilação do programa. No entanto, ainda não conseguimos treinar a rede neural, pois não conseguimos criar matrizes em primeiro lugar!

Para garantir ainda mais segurança, queremos ser capazes de criar uma matriz com valores aleatórios apenas a partir de seu tipo. Para isso, precisamos de uma forma de cruzar do mundo dos tipos para o mundo dos valores. Felizmente, nossa implementação de números naturais no nível dos tipos nos permite fazer isso com apenas algumas adições no nosso código:

```haskell
class SingNat n where
  natVal :: Proxy n -> Int

instance SingNat Z where
  natVal _ = 0

instance (SingNat n) => SingNat (S n) where
  natVal _ = 1 + natVal (Proxy :: Proxy n)

randomMatrix :: (MonadRandom m, SingNat r, SingNat c) => m (Matrix r c Double)
randomMatrix = do
  let r = natVal (Proxy :: Proxy r)
  let c = natVal (Proxy :: Proxy c)
  M <$> replicateM r (replicateM c getRandom)
```

Nossa função `randomMatrix` é capaz de criar uma matriz com valores aleatórios a partir unicamente do seu tipo. Isso é feito através da classe `SingNat` que nos permite obter o valor de um número natural `Nat` em tempo de execução baseado no seu tipo. Assim, podemos criar matrizes com valores aleatórios de forma segura, garantindo que a matriz criada tem as dimensões corretas.

Para podermos visualizar a nossa matriz, vamos criar uma instância de `Show` para a nossa matriz:

```haskell
randomMatrix :: IO (Matrix (S (S (S Z))) (S (S (S (S Z)))) Double)
```

Agora, podemos usar nossa função `randomMatrix` para criar todas as matrizes de pesos e de ativação da nossa rede neural. Com isso, podemos treinar a nossa rede e utilizá-la para realizar tarefas de aprendizado de máquina.

Conclusão

Neste texto, vimos como podemos utilizar a programação no nível dos tipos em Haskell para criar estruturas de dados com garantias de segurança e, além disso, aplicações mais complexas com utilidade prática. É possível, além dos exemplos aqui mostrados, criar estruturas de dados ainda mais complexas e garantir que todas as operações feitas com essas estruturas sejam válidas em tempo de compilação. Algumas linguagens levam isso ainda mais ao extremo, como Idris, que permite que os tipos sejam dependentes dos valores, o que aumenta ainda mais a segurança do programa.