

Если вы видите что-то необычное, просто сообщите мне.

Machine Learning in Haskell

AI and machine learning are huge topics in technology. In this series, we'll explore how Haskell's unique features as a language can be valuable in crafting better AI programs. In particular, we'll explore some advanced concepts in type safety, and apply these to the machine learning framework Tensor Flow. We'll also look at another library that uses similar ideas with Neural Networks.

- [Haskell and Tensor Flow](#)
- [Haskell, AI, and Dependent Types I](#)
- [Haskell, AI, and Dependent Types II](#)
- [Grenade and Deep Learning](#)

Haskell and Tensor Flow

AI systems are beginning to impact our lives more and more. It's a very important element to how software is being developed and will continue to be developed. But where does Haskell fit in this picture?

In this series, we'll go over the basic concepts of Tensor Flow, one of the most easier machine learning frameworks to pick. We'll first try it out in Python (the most common language for TF), and then we'll translate our code to Haskell. For some help on actually installing the Haskell Tensor Flow library so you can write your own code, make sure to download our Haskell Tensor Flow Guide!

If you're already a bit familiar with these bindings, you can move on to part 2. We'll see how we can apply some advanced type system tricks to actually make our Tensor Flow code safer!

*Note this series will not be a general introduction to the concept of machine learning. There is a fantastic series on Medium about that called Machine Learning is Fun! If you're interested in learning the basic concepts, I highly recommend you check out part 1 of that series. Many of the ideas in my own article series will be a lot clearer with that background.

TENSORS Tensor Flow is a great name because it breaks the library down into the two essential concepts. First up are tensors. These are the primary vehicle of data representation in Tensor Flow. Low-dimensional tensors are actually quite intuitive. But there comes a point when you can't really visualize what's going on, so you have to let the theoretical idea guide you.

In the world of big data, we represent everything numerically. And when you have a group of numbers, a programmer's natural instinct is to put those in an array.

```
[1.0, 2.0, 3.0, 6.7]
```

Now what do you do if you have a lot of different arrays of the same size and you want to associate them together? You make a 2-dimensional array (an array of arrays), which we also refer to as a matrix.

```
[[1.0, 2.0, 3.0, 6.7],  
 [5.0, 10.0, 3.0, 12.9],  
 [6.0, 12.0, 15.0, 13.6],  
 [7.0, 22.0, 8.0, 5.3]]
```

Most programmers are pretty familiar with these concepts. Tensors take this idea and keep extending it. What happens when you have a lot of matrices of the same size? You can group them together as an array of matrices. We could call this a three-dimensional matrix. But "tensor" is the term we'll use for this data representation in all dimensions.

Every tensor has a degree. We could start with a single number. This is a tensor of degree 0. Then a normal array is a tensor of degree 1. Then a matrix is a tensor of degree 2. Our last example would be a tensor of degree 3. And you can keep adding these on to each other, ad infinitum.

Every tensor has a shape. The shape is an array representing the dimensions of the tensor. The length of this array will be the degree of the tensor. So a number will have the empty list as its shape. An array will have a list of length 1 containing the length of the array. A matrix will have a list of length 2 containing its number of rows and columns. And so on. There are a few different ways we can represent tensors in code, but we'll get to that in a bit.

GO WITH THE FLOW

The second important concept to understand is how Tensor Flow performs computations. Machine learning generally involves simple math operations. A lot of simple math operations. Since the scale is so large, we need to perform these operations as fast as possible. And we need to use software and hardware that is optimized for these specific tasks. This necessitates having a low-level code representation of what's going on. This is easier to achieve in a language like C, instead of Haskell or Python.

We could have the bulk of our code in Haskell, but perform the math in C using a Foreign Function Interface. But these interfaces have a large overhead, so this is likely to negate most of the gains we get from using C.

Tensor Flow's solution to this problem is that we first build up a graph describing all our computations. Then once we have described that, we "run" our graph using a "session". Thus it

performs the entire language conversion process at once, so the overhead is lower.

If this sounds familiar, it's because this is how actions tend to work in Haskell (in some sense). We can, for instance, describe an IO action. And this action isn't a series of commands that we execute the moment they show up in the code. Rather, the action is a description of the operations that our program will perform at some point. It's also similar to the concept of Effectful programming.

So what does our computation graph look like? Well, each tensor is a node. Then we can make other nodes for "operations" that take tensors as input. For instance, we can "add" two tensors together, and this is another node. We'll see in our example how we build up the computational graph, and then run it.

#CODING TENSORS So at this point we should start examining how we actually create tensors in our code. We'll start by looking at how we do this in Python, since the concepts are a little easier to understand that way. There are three types of tensors we'll consider. The first are "constants". These represent a set of values that do not change. We can use these values throughout our model training process, and they'll be the same each time. Since we define the values for the tensor up front, there's no need to give any size arguments. But we will specify the datatype that we'll use for them.

```
import tensorflow as tf

node1 = tf.constant(3.0, dtype=tf.float32)
node2 = tf.constant(4.0, dtype=tf.float32)
```

Now what can we actually do with these tensors? Well for a quick sample, let's try adding them. This creates a new node in our graph that represents the addition of these two tensors. Then we can "run" that addition node to see the result. To encapsulate all our information, we'll create a "Session":

```
import tensorflow as tf

node1 = tf.constant(3.0, dtype=tf.float32)
node2 = tf.constant(4.0, dtype=tf.float32)
additionNode = tf.add(node1, node2)

sess = tf.Session()
```

```
result = sess.run(additionNode)
print result
```

```
"""
```

Output:

7.0

```
"""
```

The next type of tensors are placeholders. These are values that we change each run. Generally, we will use these for the inputs to our model. By using placeholders, we'll be able to change the input and train on different values each time. When we "run" a session, we need to assign values to each of these nodes.

We don't know the values that will go into a placeholder, but we still assign the type of data at construction. We can also assign a size if we like. So here's a quick snippet showing how we initialize placeholders. Then we can assign different values with each run of the application. Even though our placeholder tensors don't have values, we can still add them just as we could with constant tensors.

```
node1 = tf.placeholder(tf.float32)
node2 = tf.placeholder(tf.float32)
adderNode = tf.add(node1, node2)

sess = tf.Session()
result1 = sess.run(adderNode, {node1: 3, node2: 4.5 })
result2 = sess.run(adderNode, {node1: 2.7, node2: 8.9 })
print(result1)
print(result2)
```

```
"""
```

Output:

7.5

11.6

```
"""
```

The last type of tensor we'll use are variables. These are the values that will constitute our "model". Our goal is to find values for these parameters that will make our model fit the data well. We'll supply a data type, as always. In this situation, we'll also provide an initial constant value.

Normally, we'd want to use a random distribution of some kind. The tensor won't actually take on its value until we run a global variable initializer function. We'll have to create this initializer and then have our session object run it before we get going.

```
w = tf.Variable([3], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
```

Now let's use our variables to create a "model" of sorts. For this article we'll make a simple linear model. Let's create additional nodes for our input tensor and the model itself. We'll let w be the weights, and b be the "bias". This means we'll construct our final value by $w \cdot x + b$, where x is the input.

```
w = tf.Variable([3], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)
x = tf.placeholder(dtype=tf.float32)
linear_model = w * x + b
```

Now, we want to know how good our model is. So let's compare it to y , an input of our expected values. We'll take the difference, square it, and then use the `reduce_sum` library function to get our "loss". The loss measures the difference between what we want our model to represent and what it actually represents.

```
w = tf.Variable([3], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)
x = tf.placeholder(dtype=tf.float32)
linear_model = w * x + b
y = tf.placeholder(dtype=tf.float32)
squared_deltas = tf.square(linear_model - y)
loss = tf.reduce_sum(squared_deltas)
```

Each line here is a different tensor, or a new node in our graph. We'll finish up our model by using the built in `GradientDescentOptimizer` with a learning rate of 0.01. We'll set our training step as attempting to minimize the loss function.

```
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)
```

Now we'll run the session, initialize the variables, and run our training step 1000 times. We'll pass a series of inputs with their expected outputs. Let's try to learn the line $y = 5x - 1$. Our expected output y values will assume this.

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
for i in range(1000):
    sess.run(train, {x: [1, 2, 3, 4], y: [4,9,14,19]})

print(sess.run([W,b]))
```

At the end we print the weights and bias, and we see our results!

```
[array([ 4.99999475], dtype=float32), array([-0.99998516], dtype=float32)]
```

So we can see that our learned values are very close to the correct values of 5 and -1!

REPRESENTING TENSORS IN HASKELL

So now at long last, I'm going to get into some of the details of how we apply these tensor concepts in Haskell. Like strings and numbers, we can't have this one "Tensor" type in Haskell, since that type could really represent some very different concepts. For a deeper look at the tensor types we're dealing with, check out our in depth guide.

In the meantime, let's go through some simple code snippets replicating our efforts in Python. Here's how we make a few constants and add them together. Do note the "overloaded lists" extension. It allows us to represent different types with the same syntax as lists. We use this with both Shape items and Vectors:

```

{-# LANGUAGE OverloadedLists #-}

import Data.Vector (Vector)
import TensorFlow.Ops (constant, add)
import TensorFlow.Session (runSession, run)

runSimple :: IO (Vector Float)
runSimple = runSession $ do
  let node1 = constant [1] [3 :: Float]
  let node2 = constant [1] [4 :: Float]
  let additionNode = node1 `add` node2
  run additionNode

main :: IO ()
main = do
  result <- runSimple
  print result

{-
Output:
[7.0]
-}

```

We use the constant function, which takes a Shape and then the value we want. We'll create our addition node and then run it to get the output, which is a vector with a single float. We wrap everything in the runSession function. This encapsulates the initialization and running actions we saw in Python.

Now suppose we want placeholders. This is a little more complicated in Haskell. We'll be using two placeholders, as we did in Python. We'll initialize them with the placeholder function and a shape. We'll take arguments to our function for the input values. To actually pass the parameters to fill in the placeholders, we have to use what we call a "feed".

We know that our adderNode depends on two values. So we'll write our run-step as a function that takes in two "feed" values, one for each placeholder. Then we'll assign those feeds to the proper nodes using the feed function. We'll put these in a list, and pass that list as an argument to runWithFeeds. Then, we wrap up by calling our run-step on our input data. We'll have to encode the raw vectors as tensors though.


```

import TensorFlow.Core (Tensor, Value, feed, encodeTensorData)
import TensorFlow.Ops (constant, add, placeholder)
import TensorFlow.Session (runSession, run, runWithFeeds)

import Data.Vector (Vector)

runPlaceholder :: Vector Float -> Vector Float -> IO (Vector Float)
runPlaceholder input1 input2 = runSession $ do
  (node1 :: Tensor Value Float) <- placeholder [1]
  (node2 :: Tensor Value Float) <- placeholder [1]
  let adderNode = node1 `add` node2
  let runStep = \node1Feed node2Feed -> runWithFeeds
    [ feed node1 node1Feed
    , feed node2 node2Feed
    ]
    adderNode
  runStep (encodeTensorData [1] input1) (encodeTensorData [1] input2)

main :: IO ()
main = do
  result1 <- runPlaceholder [3.0] [4.5]
  result2 <- runPlaceholder [2.7] [8.9]
  print result1
  print result2

{-
Output:
[7.5]
[11.599999]
-}

```

Now we'll wrap up by going through the simple linear model scenario we already saw in Python. Once again, we'll take two vectors as our inputs. These will be the values we try to match. Next, we'll use the `initializedVariable` function to get our variables. We don't need to call a global variable initializer. But this does affect the state of the session. Notice that we pull it out of the monad context, rather than using `let`. (We also did for placeholders.)

```

import TensorFlow.Core (Tensor, Value, feed, encodeTensorData, Scalar(..))
import TensorFlow.Ops (constant, add, placeholder, sub, reduceSum, mul)

```

```

import TensorFlow.GenOps.Core (square)
import TensorFlow.Variable (readValue, initializedVariable, Variable)
import TensorFlow.Session (runSession, run, runWithFeeds)
import TensorFlow.Minimize (gradientDescent, minimizeWith)

import Control.Monad (replicateM_)
import qualified Data.Vector as Vector
import Data.Vector (Vector)

runVariable :: Vector Float -> Vector Float -> IO (Float, Float)
runVariable xInput yInput = runSession $ do
  let xSize = fromIntegral $ Vector.length xInput
  let ySize = fromIntegral $ Vector.length yInput
  (w :: Variable Float) <- initializedVariable 3
  (b :: Variable Float) <- initializedVariable 1
  ...

```

Next, we'll make our placeholders and linear model. Then we'll calculate our loss function in much the same way we did before. Then we'll use the same feed trick to get our placeholders plugged in.

```

runVariable :: Vector Float -> Vector Float -> IO (Float, Float)
...
(x :: Tensor Value Float) <- placeholder [xSize]
let linear_model = ((readValue w) `mul` x) `add` (readValue b)
(y :: Tensor Value Float) <- placeholder [ySize]
let square_deltas = square (linear_model `sub` y)
let loss = reduceSum square_deltas
trainStep <- minimizeWith (gradientDescent 0.01) loss [w,b]
let trainWithFeeds = \xF yF -> runWithFeeds
  [ feed x xF
  , feed y yF
  ]
  trainStep
...

```

Finally, we'll run our training step 1000 times on our input data. Then we'll run our model one more time to pull out the values of our weights and bias. Then we're done!

```

runVariable :: Vector Float -> Vector Float -> IO (Float, Float)

...
replicateM_ 1000
  (trainWithFeeds (encodeTensorData [xSize] xInput) (encodeTensorData [ySize] yInput))
  (Scalar w_learned, Scalar b_learned) <- run (readValue w, readValue b)
return (w_learned, b_learned)

main :: IO ()
main = do
  results <- runVariable [1.0, 2.0, 3.0, 4.0] [4.0, 9.0, 14.0, 19.0]
  print results

{-
Output:
(4.9999948,-0.99998516)
-}

```

CONCLUSION

Hopefully this article gave you a taste of some of the possibilities of Tensor Flow in Haskell. We saw a quick introduction to the fundamentals of Tensor Flow. We saw three different kinds of tensors. We then saw code examples both in Python and in Haskell. Finally, we went over a very quick example of a simple linear model and saw how we could learn values to fit that model.

Now that we've got the basics down, we're going to spice things up a lot! In part 2 we'll explore the question of program safety. We'll see that our Haskell code is not necessarily any better than the Python code! But then we'll see how we can use some awesome dependent type techniques to change this!

If you want more details on running this Tensor Flow code yourself, you should check out Haskell Tensor Flow Guide! It will walk you through using the Tensor Flow library as a dependency and getting a basic model running!

Haskell, AI, and Dependent Types I

I often argue that Haskell is a safe language. There are a lot of errors we will catch at compile time, rather than runtime. Runtime errors can often be catastrophic to a system, so being able to reduce these is paramount. This is especially true when programming an autonomous car or drone. These objects will be out in the real world where they can hurt people if they malfunction.

So let's take a look back at some of the code wrote in part 1 of this series. Is the Haskell version actually any safer than the Python version? We'll find the answer is, well, not so much. It's hard to verify certain properties about code. But the facilities for making this code safer do exist in Haskell! In this part as well as part 3, we'll do some serious hacking with dependent types. We'll be able to prove some of these difficult properties of AI programs at compile time!

The next two parts will focus on dependent type programming. This is a difficult topic, so don't worry if you can't follow all the code examples completely. The main idea of making our machine learning code safer is what's important! So without further ado, let's rewind to the beginning to see where runtime issues can appear.

If you want to play with this code yourself, check out the dependent shapes branch on my Github repository! All the code for this article is in `DepShape.hs` Though if you want to get the code to run, you'll probably also need to get Haskell Tensor Flow working. Download our Haskell Tensor Flow Guide for instructions on that!

ISSUES WITH PYTHON Python, as an interpreted language, is definitely subject to runtime bugs. As I was first learning Tensor Flow, I came across a lot of these that were quite common. The two that stood out to me most were placeholder failures and dimension mismatches. For instance, let's think back to one of the first examples. Our code will have a couple of placeholders, and we submit values for those when we run the session:

```
node1 = tf.placeholder(tf.float32)
node2 = tf.placeholder(tf.float32)
```

```
adderNode = tf.add(node1, node2)
sess = tf.Session()
result1 = sess.run(adderNode, {node1: 3, node2: 4.5 })
```

But there's nothing stopping us from trying to run the session without submitting values. This will result in a runtime crash:

```
...
sess = tf.Session()
result1 = sess.run(adderNode)
print(result1)
...
```

Terminal Output:

```
# InvalidArgumentError (see above for traceback): You must feed a value for placeholder tensor 'Placeholder'
with dtype float
# [[Node: Placeholder = Placeholder[dtype=DT_FLOAT, shape=[],
_device="/job:localhost/replica:0/task:0/cpu:0"]()]]]
```

Another issue that came up from time to time was dimension mismatches. Certain operations need certain relationships between the dimensions of the tensors. For instance, you can't add two vectors with different lengths:

```
node1 = tf.constant([3.0, 4.0, 5.0], dtype=tf.float32)
node2 = tf.constant([4.0, 16.0], dtype=tf.float32)
additionNode = tf.add(node1, node2)

sess = tf.Session()
result = sess.run(additionNode)
print(result)
```

...

Terminal Output:

```
ValueError: Dimensions must be equal, but are 3 and 2 for 'Add' (op: 'Add') with input shapes: [3], [2].
```

Again, we get a runtime crash. These seem like the kinds of problems we can solve at compile time.

DOES HASKELL SOLVE THESE ISSUES?

But anyone who takes a close look at the Haskell code I've written so far can see that it doesn't solve these issues! Here's a review of our basic placeholder example:

```
runPlaceholder :: Vector Float -> Vector Float -> IO (Vector Float)
runPlaceholder input1 input2 = runSession $ do
  (node1 :: Tensor Value Float) <- placeholder [1]
  (node2 :: Tensor Value Float) <- placeholder [1]
  let adderNode = node1 `add` node2
  let runStep = \node1Feed node2Feed -> runWithFeeds
    [ feed node1 node1Feed
    , feed node2 node2Feed
    ]
    adderNode
  runStep (encodeTensorData [1] input1) (encodeTensorData [1] input2)
```

Notice how the `runWithFeeds` function takes a list of `Feed` objects. The code would still compile fine if we supplied the empty list. Then it would face a fate no better than our Python code:

```
...
let runStep = \node1Feed node2Feed -> runWithFeeds [] adderNode
...
```

Terminal Output:

```
TensorFlowException TF_INVALID_ARGUMENT "You must feed a value for placeholder tensor 'Placeholder_1' with
dtype float and shape [1]\n\t [[Node: Placeholder_1 = Placeholder[dtype=DT_FLOAT, shape=[1],
_device=\"/job:localhost/replica:0/task:0/cpu:0\"]()]]"
```

For the second example of dimensionality, we can also make this mistake in Haskell. The following code compiles and will crash at runtime:

```
runSimple :: IO (Vector Float)
runSimple = runSession $ do
  let node1 = constant [3] [3 :: Float, 4, 5]
  let node2 = constant [2] [4 :: Float, 5]
  let additionNode = node1 `add` node2
  run additionNode
...

-- Terminal Output:
-- TensorFlowException TF_INVALID_ARGUMENT "Incompatible shapes: [3] vs. [2]\n\t [[Node: Add_2 =
Add[T=DT_FLOAT, _device=\"/job:localhost/replica:0/task:0/cpu:0\"](Const_0, Const_1)]]"
```

At an even more basic level, we don't even have to tell the truth about the shape of our vectors! We can give a bogus shape value and it will still compile!

```
let node1 = constant [3, 2, 3] [3 :: Float, 4, 5]
...

# Terminal Output:
# invalid tensor length: expected 18 got 3
# CallStack (from HasCallStack):
#   error, called at src/TensorFlow/Ops.hs:299:23 in tensorflow-ops-0.1.0.0-
EWsy8DQdciaL8o6yb2fUKR:TensorFlow.Ops
```

CAN WE DO BETTER?

When trying to solve these, we could write wrappers around every operation. Functions like `add` and `matMul` could return `Maybe` values. But this would be clunky. We could take this same step in Python. Granted, monads would allow the Haskell version to compose better. But it would be nicer if we could check our errors all at once, up front.

If we're willing to dig quite a bit deeper, we can solve these problems! In the rest of this part, we'll explore using dependent types to ensure dimensions are always correct. Getting placeholders right

turns out to be a little more complicated though! So we'll save that for part 4.

CHECKING DIMENSIONS

Currently, the Tensor Types we've been dealing with have no type safety on the dimensions. Tensor Flow doesn't provide this information when interacting with the C library. So it's impossible to enforce it at a low level. But this doesn't stop us from writing wrappers that allow us to solve this.

To write these wrappers, we're going to need to dive into dependent types. I'll give a high level overview of what's going on. But for some details on the basics, you should check out this tutorial . I'll also give a shout-out to Renzo Carbonara, author of the Exinst library and other great Haskell things. He helped me a lot in crossing a couple big knowledge gaps for implementing dependent types.

INTRO TO DEPENDENT TYPES: SIZED VECTORS The simplest example for introducing dependent types is the idea of sized vectors. If you read the tutorial above, you'll see how they're implemented from scratch. A normal vector has a single type parameter, referring to what type of item the vector contains. A sized vector has an extra type parameter, and this type refers to the size of the vector. For instance, the following are valid sized vector types:

```
import Data.Vector.Sized (Vector, fromList)

vectorWith2 :: Vector 2 Int64
...
vectorWith6 :: Vector 6 Float
...
```

In the first type signature, 2 does not refer to the term 2. It refers to the type 2. That is, we've taken the term and promoted it to a type which has only a single value. The mechanics of how this works are confusing, but here's the result. We can try to convert normal vectors to sized vectors. But the operation will fail if we don't match up the size.

```
import Data.Vector.Sized (Vector, fromList)
import GHC.TypeLits (KnownNat)
```



```
-- fromList :: (KnownNat n) => [a] -> Maybe (Vector n a)

-- This results in a "Just" value!
success :: Maybe (Vector 2 Int64)
success = fromList [5,6]

-- The sizes don't match, so we'll get "Nothing"!
failure :: Maybe (Vector 2 Int64)
failure = fromList [3,1,5]
```

The KnownNat constraint allows us to specify that the type n refers to a single natural number. So now we can assign a type signature that encapsulates the size of the list.

A "SAFE" SHAPE TYPE

Now that we have a very basic understanding of dependent types, let's come up with a gameplan for Tensor Flow. The first step will be to make a new type that puts the shape into the type signature. We'll make a SafeShape type that mimics the sized vector type. Instead of storing a single number as the type, it will store the full list of dimensions. We want to create an API something like this:

```
-- fromShape :: Shape -> Maybe (SafeShape s)

-- Results in a "Just" value
goodShape :: Maybe (SafeShape '[2, 2])
goodShape = fromShape (Shape [2,2])

-- Results in Nothing
badShape :: Maybe (SafeShape '[2,2])
badShape = fromShape (Shape [3,3,2])
```

So to do this, we first define the SafeShape type. This follows the example of sized vectors. See the appendix below for compiler extensions and imports used throughout this article. In particular, you want GADTs and DataKinds.

```

data SafeShape (s :: [Nat]) where
  NilShape :: SafeShape '[]
  (:-) :: KnownNat m => Proxy m -> SafeShape s -> SafeShape (m ': s)

infixr 5 :-

```

Now we can define the `toShape` function. This will take our `SafeShape` and turn it into a normal `Shape` using proxies.

```

toShape :: SafeShape s -> Shape
toShape NilShape = Shape []
toShape ((pm :: Proxy m) :- s) = Shape (fromInteger (natVal pm) : s')
  where
    (Shape s') = toShape s

```

Now for the reverse direction, we first have to make a class `MkSafeShape`. This class encapsulates all the types that we can turn into the `SafeShape` type. We'll define instances of this class for all lists of naturals.

```

class MkSafeShape (s :: [Nat]) where
  mkSafeShape :: SafeShape s
instance MkSafeShape '[] where
  mkSafeShape = NilShape
instance (MkSafeShape s, KnownNat m) => MkSafeShape (m ': s) where
  mkSafeShape = Proxy :- mkSafeShape

```

Now we can define our `fromShape` function using the `MkSafeShape` class. To check if it works, we'll compare the resulting shape to the input shape and make sure they're equal. Note this requires us to define a simple instance of `Eq Shape`.

```

instance Eq Shape where
  (==) (Shape s) (Shape r) = s == r

fromShape :: forall s. MkSafeShape s => Shape -> Maybe (SafeShape s)
fromShape shape = if toShape myShape == shape
  then Just myShape
  else Nothing
  where
    myShape = mkSafeShape :: SafeShape s

```

Now that we've done this for Shape, we can create a similar type for Tensor that will store the shape as a type parameter.

```
data SafeTensor v a (s :: [Nat]) where
  SafeTensor :: (TensorType a) => Tensor v a -> SafeTensor v a s
```

USING OUR SAFE TYPES

So what has all this gotten us? Our next goal is to create a `safeConstant` function. This will let us create a `SafeTensor` wrapping a constant tensor and storing the shape. Remember, `constant` takes a shape and a vector without ensuring correlation between them. We want something like this:

```
safeConstant :: (TensorType a) => Vector n a -> SafeShape s -> SafeTensor Build a s
safeConstant elems shp = SafeTensor $ constant (toShape shp) (toList elems)
```

This will attach the given shape to the tensor. But there's one piece missing. We also want to create a connection between the number of input elements and the shape. So something with shape `[3,3,2]` should force you to input a vector of length 18. And right now, there is no constraint between `n` and `s`.

We'll add this with a type family called `ShapeProduct`. The instances will state that the correct natural type for a given list of naturals is the product of them. We define the second instance with recursion, so we'll need `UndecidableInstances`.

```
type family ShapeProduct (s :: [Nat]) :: Nat
type instance ShapeProduct '[] = 1
type instance ShapeProduct (m ': s) = m * ShapeProduct s
```

Now we're almost done with this part! We can fix our `safeConstant` function by adding a constraint on the `ShapeProduct` between `s` and `n`.

```
safeConstant :: (TensorType a, ShapeProduct s ~ n) => Vector n a -> SafeShape s -> SafeTensor Build a s
safeConstant elems shp = SafeTensor $ constant (toShape shp) (toList elems)
```

Now we can write out a simple use of our `safeConstant` function as follows:

```

main :: IO (VN.Vector Int64)
main = runSession $ do
  let (shape1 :: SafeShape '[2,2]) = fromJust $ fromShape (Shape [2,2])
  let (elems1 :: Vector 4 Int64) = fromJust $ fromList [1,2,3,4]
  let (constant1 :: SafeTensor Build Int64 '[2,2]) = safeConstant elems1 shape1
  let (SafeTensor t) = constant1
  run t

```

We're using `fromJust` as a shortcut here. But in a real program you would read your initial tensors in and check them as `Maybe` values. There's still the possibility for runtime failures. But this system has a couple advantages. First, it won't crash. We'll have the opportunity to handle it gracefully. Second, we do all the error checking up front. Once we've assigned types to everything, all the failure cases should be covered.

Going back to the last example, let's change something. For instance, we could make our vector have length 3 instead of 4. We'll now get a compile error!

```

main :: IO (VN.Vector Int64)
main = runSession $ do
  let (shape1 :: SafeShape '[2,2]) = fromJust $ fromShape (Shape [2,2])
  let (elems1 :: Vector 3 Int64) = fromJust $ fromList [1,2,3]
  let (constant1 :: SafeTensor Build Int64 '[2,2]) = safeConstant elems1 shape1
  let (SafeTensor t) = constant1
  run t

```

...

- Couldn't match type '4' with '3'
arising from a use of 'safeConstant'
- In the expression: `safeConstant elems1 shape1`
In a pattern binding:
`(constant1 :: SafeTensor Build Int64 '[2, 2])`
`= safeConstant elems1 shape1`

ADDING TYPE SAFE OPERATIONS

Now that we've attached shape information to our tensors, we can define safer math operations. It's easy to write a safe addition function that ensures that the tensors have the same shape:

```
safeAdd :: (TensorType a, a /= Bool) => SafeTensor Build a s -> SafeTensor Build a s -> SafeTensor Build a s
safeAdd (SafeTensor t1) (SafeTensor t2) = SafeTensor (t1 `add` t2)
```

Here's a similar matrix multiplication function. It ensures we have 2-dimensional shapes and that the dimensions work out. Notice the two tensors share the n dimension. It must be the column dimension of the first tensor and the row dimension of the second tensor:

```
safeMatMul :: (TensorType a, a /= Bool, a /= Int8, a /= Int16, a /= Int64, a /= Word8, a /= ByteString)
=> SafeTensor Build a '[i,n] -> SafeTensor Build a '[n,o] -> SafeTensor Build a '[i,o]
safeMatMul (SafeTensor t1) (SafeTensor t2) = SafeTensor (t1 `matMul` t2)
```

Here are these functions in action:

```
main2 :: IO (VN.Vector Float)
main2 = runSession $ do
  let (shape1 :: SafeShape '[4,3]) = fromJust $ fromShape (Shape [4,3])
  let (shape2 :: SafeShape '[3,2]) = fromJust $ fromShape (Shape [3,2])
  let (shape3 :: SafeShape '[4,2]) = fromJust $ fromShape (Shape [4,2])
  let (elems1 :: Vector 12 Float) = fromJust $ fromList [1,2,3,4,1,2,3,4,1,2,3,4]
  let (elems2 :: Vector 6 Float) = fromJust $ fromList [5,6,7,8,9,10]
  let (elems3 :: Vector 8 Float) = fromJust $ fromList [11,12,13,14,15,16,17,18]
  let (constant1 :: SafeTensor Build Float '[4,3]) = safeConstant elems1 shape1
  let (constant2 :: SafeTensor Build Float '[3,2]) = safeConstant elems2 shape2
  let (constant3 :: SafeTensor Build Float '[4,2]) = safeConstant elems3 shape3
  let (multTensor :: SafeTensor Build Float '[4,2]) = constant1 `safeMatMul` constant2
  let (addTensor :: SafeTensor Build Float '[4,2]) = multTensor `safeAdd` constant3
  let (SafeTensor finalTensor) = addTensor
  run finalTensor
```

And of course we'll get compile errors if we use incorrect dimensions anywhere. Let's say we change `multTensor` to use `[4,3]` as its type:

```
• Couldn't match type '2' with '3'
  Expected type: SafeTensor Build Float '[4, 3]
  Actual type: SafeTensor Build Float '[4, 2]
• In the expression: constant1 `safeMatMul` constant2
...
• Couldn't match type '3' with '2'
  Expected type: SafeTensor Build Float '[4, 2]
  Actual type: SafeTensor Build Float '[4, 3]
• In the expression: multTensor `safeAdd` constant3
...
• Couldn't match type '2' with '3'
  Expected type: SafeTensor Build Float '[4, 3]
  Actual type: SafeTensor Build Float '[4, 2]
• In the second argument of 'safeAdd', namely 'constant3'
```

CONCLUSION

In this exercise we got deep into the weeds of one of the most difficult topics in Haskell. Dependent types will make your head spin at first. But we saw a concrete example of how they can allow us to detect problematic code at compile time. They are a form of documentation that also enables us to verify that our code is correct in certain ways.

Types do not replace tests (especially behavioral tests). But in this instance there are at least a few different test cases we don't need to worry about too much. In part 4, we'll see how we can apply these principles to verifying placeholders.

If you want to learn more about the nuts and bolts of using Haskell Tensor Flow, you should check out our Tensor Flow Guide. It will guide you through the basics of adding Tensor Flow to a simple Stack project.

APPENDIX: EXTENSIONS AND IMPORTS

```
{-# LANGUAGE GADTs          #-}  
{-# LANGUAGE DataKinds      #-}  
{-# LANGUAGE KindSignatures  #-}  
{-# LANGUAGE TypeOperators   #-}  
{-# LANGUAGE ScopedTypeVariables #-}  
{-# LANGUAGE TypeFamilies    #-}  
{-# LANGUAGE UndecidableInstances #-}  
  
import      Data.ByteString (ByteString)  
import      Data.Constraint (Constraint)  
import      Data.Int (Int64, Int8, Int16)  
import      Data.Maybe (fromJust)  
import      Data.Proxy (Proxy(..))  
import qualified Data.Vector as VN  
import      Data.Vector.Sized (Vector(..), toList, fromList)  
import      Data.Word (Word8)  
import      GHC.TypeLits (Nat, KnownNat, natVal)  
import      GHC.TypeLits  
  
import      TensorFlow.Core  
import      TensorFlow.Core (Shape(..), TensorType, Tensor, Build)  
import      TensorFlow.Ops (constant, add, matMul)  
import      TensorFlow.Session (runSession, run)
```

Haskell, AI, and Dependent Types II

In part 2 we dove into the world of dependent types. We linked tensors with their shapes at the type level. This gave our program some extra type safety and allowed us to avoid certain runtime errors.

In this part, we're going to solve another runtime conundrum: missing placeholders. We'll add some more dependent type machinery to ensure we've plugged in all the necessary placeholders! But we'll see this is not as straightforward as shapes.

To follow along with the code in this article, take a look at this branch on my Haskell Tensor Flow Github repository. All the code for this article is in `DepShape.hs`. As usual, I've listed the necessary compiler extensions and imports at the bottom of this article. If you want to run the code yourself, you'll have to get Haskell and Tensor Flow running first. Take a look at our Haskell Tensor Flow guide for that!

If you want to see a cleaner version of dependent types with machine learning, you should check out the last part of this series! We'll look at Grenade, a library that uses dependent types to force your Neural Networks to have the right structure!

PLACEHOLDER REVIEW

To start, let's remind ourselves what placeholders are in Tensor Flow and how we use them. Placeholders represent tensors that can have different values on different application runs. This is often the case when we're training on different samples of data. Here's our very simple example in Python. We'll create a couple placeholder tensors by providing their shapes and no values. Then when we actually run the session, we'll provide a value for each of those tensors.


```

node1 = tf.placeholder(tf.float32)
node2 = tf.placeholder(tf.float32)
adderNode = tf.add(node1, node2)
sess = tf.Session()
result1 = sess.run(adderNode, {node1: 3, node2: 4.5 })

```

The weakness here is that there's nothing forcing us to provide values for those tensors! We could try running our program without them and we'll get a runtime crash:

```

...
sess = tf.Session()
result1 = sess.run(adderNode)
print(result1)
...

# Terminal Output:

# InvalidArgumentError (see above for traceback): You must feed a value for placeholder tensor 'Placeholder'
with dtype float
#   [[Node: Placeholder = Placeholder[dtype=DT_FLOAT, shape=[],
_device="/job:localhost/replica:0/task:0/cpu:0"]()]]]

```

Unfortunately, the Haskell Tensor Flow library doesn't actually do any better here. When we want to fill in placeholders, we provide a list of “feeds”. But our program will still compile even if we pass an empty list! We'll encounter similar runtime errors:

```

(node1 :: Tensor Value Float) <- placeholder [1]
(node2 :: Tensor Value Float) <- placeholder [1]
let adderNode = node1 `add` node2
let runStep = \node1Feed node2Feed -> runWithFeeds [] adderNode
runStep (encodeTensorData [1] input1) (encodeTensorData [1] input2)
...

-- Terminal Output:

-- TensorFlowException TF_INVALID_ARGUMENT "You must feed a value for placeholder tensor 'Placeholder_1'
with dtype float and shape [1]\n\t [[Node: Placeholder_1 = Placeholder[dtype=DT_FLOAT, shape=[1],
_device="/job:localhost/replica:0/task:0/cpu:0"]()]]]"

```

One solution, which you can explore here, is to bury the call to `runWithFeeds` within our neural network API. We only provide a `Model` object. This model object forces us to provide the expected input and output tensors. So anyone using our model wouldn't make a manual `runWithFeeds` call.

```
data Model = Model
  { train :: TensorData Float
    -> TensorData Int64
    -> Session ()
  , errorRate :: TensorData Float
    -> TensorData Int64
    -> SummaryTensor
    -> Session (Float, ByteString)
  }
```

This isn't a bad solution! But it's interesting to see how we can push the envelope with dependent types, so let's try that!

ADDING MORE “SAFE” TYPES

The first step we'll take is to augment Tensor Flow's `TensorData` type. We'll want it to have shape information like `SafeTensor` and `SafeShape`. But we'll also attach a name to each piece of data. This will allow us to identify which tensor to substitute the data in for. At the type level, we refer to this name as a `Symbol`.

```
data SafeTensorData a (n :: Symbol) (s :: [Nat]) where
  SafeTensorData :: (TensorType a) => TensorData a -> SafeTensorData a n s
```

Next, we'll need to make some changes to our `SafeTensor` type. First, each `SafeTensor` will get a new type parameter. This parameter refers to a mapping of names (symbols) to shapes (which are still lists of naturals). We'll call this a placeholder list. So each tensor will have type-level information for the placeholders it depends on. Each different placeholder has a name and a shape.

```
data SafeTensor v a (s :: [Nat]) (p :: [(Symbol, [Nat])]) where
  SafeTensor :: (TensorType a) => Tensor v a -> SafeTensor v a s p
```

Now, recall when we substituted for placeholders, we used a list of feeds. But this list had no information about the names or dimensions of its feeds. Let's create a new type containing the different elements we need for our feeds. It should also contain the correct type information about the placeholder list. The first step of to define the type so that it has the list of placeholders it contains, like the `SafeTensor`.

```
data FeedList (pl :: [(Symbol, [Nat])]) where
```

This structure will look like a linked list, like our `SafeShape`. Thus we'll start by defining an “empty” constructor:

```
data FeedList (pl :: [(Symbol, [Nat])]) where
  EmptyFeedList :: FeedList '[]
```

Now we'll add a “Cons”-like constructor by creating yet another type operator `:-:`. Each “piece” of our linked list will contain two different items. First, the tensor we are substituting for. Next, it will have the data we'll be using for the substitution. We can use type parameters to force their shapes and data types to match. Then we need the resulting placeholder type. We have to append the type-tuple containing the symbol and shape to the previous list. This completes our definition.

```
data FeedList (pl :: [(Symbol, [Nat])]) where
  EmptyFeedList :: FeedList '[]
  (:-:) :: (KnownSymbol n)
    => (SafeTensor Value a s p, SafeTensorData a n s)
    -> FeedList pl
    -> FeedList ( '(n, s) ': pl)

infixr 5 :-:
```

Note that we force the tensor to be a `Value` tensor. We can only substitute data for rendered tensors, hence this restriction. Let's add a quick `safeRender` so we can render our `SafeTensor` items.

```
safeRender :: (MonadBuild m) => SafeTensor Build a s pl -> m (SafeTensor Value a s pl)
safeRender (SafeTensor t1) = do
  t2 <- render t1
  return $ SafeTensor t2
```

MAKING A PLACEHOLDER

Now we can write our `safePlaceholder` function. We'll add a `KnownSymbol` as a type constraint. Then we'll take a `SafeShape` to give ourselves the type information for the shape. The result is a new tensor that maps the symbol and the shape in the placeholder list.

```
safePlaceholder :: (MonadBuild m, TensorType a, KnownSymbol sym) =>
  SafeShape s -> m (SafeTensor Value a s '[ '(sym, s)])
safePlaceholder shp = do
  pl <- placeholder (toShape shp)
  return $ SafeTensor pl
```

This looks a little crazy, and it kind've is! But we've now created a tensor that stores its own placeholder information at the type level!

UPDATING OLD CODE

Now that we've done this, we're also going to have to update some of our older code. The first part of this is pretty straightforward. We'll need to change `safeConstant` so that it has the type information. It will have an empty list for the placeholders.

```
safeConstant :: (TensorType a, ShapeProduct s ~ n) =>
  Vector n a -> SafeShape s -> SafeTensor Build a s '[]
safeConstant elems shp = SafeTensor (constant (toShape shp) (toList elems))
```

Our mathematical operations will be a bit more tricky though. Consider adding two arbitrary tensors. They may share placeholder dependencies but may not. What should be the placeholder type for the resulting tensor? Obviously the union of the two placeholder maps of the input tensors! Luckily for us, we can use `Union` from the `type-list` library to represent this concept.

```
safeAdd :: (TensorType a, a /= Bool, TensorKind v)
=> SafeTensor v a s p1
-> SafeTensor v a s p2
-> SafeTensor Build a s (Union p1 p2)
```

```
safeAdd (SafeTensor t1) (SafeTensor t2) = SafeTensor (t1 `add` t2)
```

We'll make the same update with matrix multiplication:

```
safeMatMul :: (TensorType a, a /= Bool, a /= Int8, a /= Int16,  
              a /= Int64, a /= Word8, a /= ByteString, TensorKind v)  
=> SafeTensor v a '[i,n] p1 -> SafeTensor v a '[n,o] p2 -> SafeTensor Build a '[i,o] (Union p1 p2)  
safeMatMul (SafeTensor t1) (SafeTensor t2) = SafeTensor (t1 `matMul` t2)
```

RUNNING WITH PLACEHOLDERS

Now we have all the information we need to write our `safeRun` function. This will take a `SafeTensor`, and it will also take a `FeedList` with the same placeholder type. Remember, a `FeedList` contains a series of `SafeTensorData` items. They must match up symbol-for-symbol and shape-for-shape with the placeholders within the `SafeTensor`. Let's look at the type signature:

```
safeRun :: (TensorType a, Fetchable (Tensor v a) r) =>  
          FeedList pl -> SafeTensor v a s pl -> Session r
```

The `Fetchable` constraint enforces that we can actually get the “result” `r` out of our tensor. For instance, we can “fetch” a vector of floats out of a tensor that uses `Float` as its underlying value.

We'll next define a tail-recursive helper function to build the vanilla “list of feeds” out of our `FeedList`. Through pattern matching, we can pick out the tensor to substitute for and the data we're using. We can combine these into a feed and append to the growing list:

```
safeRun = ...  
where  
  buildFeedList :: FeedList ss -> [Feed] -> [Feed]  
  buildFeedList EmptyFeedList accum = accum  
  buildFeedList ((SafeTensor tensor_, SafeTensorData data_) :--: rest) accum =  
    buildFeedList rest ((feed tensor_ data_) : accum)
```

Now all we have to do to finish up is call the normal `runWithFeeds` function with the list we've created!

```
safeRun :: (TensorType a, Fetchable (Tensor v a) r) =>
  FeedList pl -> SafeTensor v a s pl -> Session r
safeRun feeds (SafeTensor finalTensor) = runWithFeeds (buildFeedList feeds []) finalTensor
  where
  ...
```

And here's what it looks like to use this in practice with our simple example. Notice the type signatures do get a little cumbersome. The signatures we place on the initial placeholder tensors are necessary. Otherwise the compiler wouldn't know what label we're giving them! The signature containing the union of the types is unnecessary. We can remove it if we want and let type inference do its work.

```
main3 :: IO (VN.Vector Float)
main3 = runSession $ do
  let (shape1 :: SafeShape '[2,2]) = fromJust $ fromShape (Shape [2,2])
  (a :: SafeTensor Value Float '[2,2] ['("a", '[2,2])]) <- safePlaceholder shape1
  (b :: SafeTensor Value Float '[2,2] ['("b", '[2,2])]) <- safePlaceholder shape1
  let result = a `safeAdd` b
  (result_ :: SafeTensor Value Float '[2,2] ['("b", '[2,2]), ("a", '[2,2])]) <- safeRender result
  let (feedA :: Vector 4 Float) = fromJust $ fromList [1,2,3,4]
  let (feedB :: Vector 4 Float) = fromJust $ fromList [5,6,7,8]
  let fullFeedList = (b, safeEncodeTensorData shape1 feedB) :--:
    (a, safeEncodeTensorData shape1 feedA) :--:
    EmptyFeedList
  safeRun fullFeedList result_

{- It runs!
[6.0,8.0,10.0,12.0]
-}
```

Now suppose we make some mistakes with our types. Here we'll take out the "A" feed from our feed list:

```
-- Let's take out Feed A!
main = ...
  let fullFeedList = (b, safeEncodeTensorData shape1 feedB) :--:
```

```

EmptyFeedList
safeRun fullFeedList result_

-- {- Compiler Error!
-- • Couldn't match type '['('a', '[2, 2])]' with '['[
--   Expected type: SafeTensor Value Float '[2, 2] '['('b', '[2, 2])]
--   Actual type: SafeTensor
--           Value Float '[2, 2] '['('b', '[2, 2]), ('a', '[2, 2])]'
-- }

```

Here's what happens when we try to substitute a vector with the wrong size. It will identify that we have the wrong number of elements!

```

main = ...
-- Wrong Size!
let (feedA :: Vector 8 Float) = fromJust $ fromList [1,2,3,4,5,6,7,8]
let (feedB :: Vector 4 Float) = fromJust $ fromList [5,6,7,8]
let fullFeedList = (b, safeEncodeTensorData shape1 feedB) :--:
    (a, safeEncodeTensorData shape1 feedA) :--:
    EmptyFeedList
safeRun fullFeedList result_

{- Compiler Error!
Couldn't match type '4' with '8'
    arising from a use of 'safeEncodeTensorData'
-}

```

CONCLUSION: PROS AND CONS

So let's take a step back and look at what we've constructed here. We've managed to provide ourselves with some pretty cool compile time guarantees. We've also added de-facto documentation to our code. Anyone familiar with the codebase can tell at a glance what placeholders we need for each tensor. It's a lot harder now to write incorrect code. There are still error conditions of course. But if we're smart we can write our code to deal with these all upfront.

That way we can fail gracefully instead of throwing a random run-time crash somewhere.

But there are drawbacks. Imagine being a Haskell novice and walking into this codebase. You'll have no real clue what's going on. The types are very cumbersome after a while, so continuing to write them down gets very tedious. Though as I mentioned, type inference can deal with a lot of that. But if you don't track them, the type union can be finicky about the ordering of your placeholders. We could fix this with another type family though.

All these factors could present a real drag on development. But then again, tracking down run-time errors can also do this. Tensor Flow's error messages can still be a little cryptic. This can make it hard to find root causes.

Since I'm still a novice with dependent types, this code was a little messy. In the fourth and final part of this series, we'll take a look at a more polished library that uses dependent types for neural networks. We'll see how the Grenade library allows us to specify a learning system in just a few lines of code!

If you to try out Tensor Flow, download our Tensor Flow Guide! It will walk you through incorporating the library into a Stack project!

APPENDIX: COMPILER EXTENSIONS AND IMPORTS

```
{-# LANGUAGE GADTs          #-}  
{-# LANGUAGE DataKinds      #-}  
{-# LANGUAGE KindSignatures #-}  
{-# LANGUAGE TypeOperators   #-}  
{-# LANGUAGE ScopedTypeVariables #-}  
{-# LANGUAGE TypeFamilies    #-}  
{-# LANGUAGE FlexibleContexts #-}  
{-# LANGUAGE UndecidableInstances #-}  
  
import      Data.ByteString (ByteString)  
import      Data.Int (Int64, Int8, Int16)
```



```
import Data.Maybe (fromJust)
import Data.Proxy (Proxy(..))
import Data.Type.List (Union)
import qualified Data.Vector as VN
import Data.Vector.Sized (Vector, toList, fromList)
import Data.Word (Word8)
import GHC.TypeLits (Nat, KnownNat, natVal)
import GHC.TypeLits

import TensorFlow.Core
import TensorFlow.Core (Shape(..), TensorType, Tensor, Build)
import TensorFlow.Ops (constant, add, matMul, placeholder)
import TensorFlow.Session (runSession, run)
import TensorFlow.Tensor (TensorKind)
```

Grenade and Deep Learning

In part 2 and part 3 of this series, we explored some of the most complex topics in Haskell. We examined potential runtime failures that can occur when using Tensor Flow. These included mismatched dimensions and missing placeholders. In an ideal world, we would catch these issues at compile time instead. At its current stage, the Haskell Tensor Flow library doesn't support that. But we demonstrated that it is possible to do this by using dependent types.

If you want to explore coding with the Haskell Tensor Flow library yourself, make sure you download our Guide. It'll give you a lot of tips on what dependencies you need and how to install everything!

Now, I'm still very much of a novice at dependent types, so the solutions I presented were rather clunky. In this final part, I'll show a better example of this concept from a different library. The Grenade library uses dependent types everywhere. It allows us to build verifiably-valid neural networks with extreme concision. Since it's so easy to build a larger network, Grenade can be a powerful tool for deep learning! So let's dive in and see what it's all about! The code for this part is on the grenade branch of our Github repository.

SHAPES AND LAYERS

The first thing to learn with this library is the twin concepts of Shapes and Layers. Shapes are best compared to tensors from Tensor Flow, except that they exist at the type level. In Tensor Flow we could build tensors with arbitrary dimensions. Grenade currently only supports up to three dimensions. So the different shape types either start with D1, D2, or D3, depending on the dimensionality of the shape. Then each of these type constructors take a set of natural number parameters. So the following are all valid “Shape” types within Grenade:

```
D1 5  
D2 4 12  
D3 8 10 2
```

The first represents a vector with 5 elements. The second represents a matrix with 4 rows and 12 columns. And the third represents an 8x10x2 matrix (or tensor, if you like). The different numbers represent those values at the type level, not the term level. If this seems confusing, here's a good tutorial that goes into more depth about the basics of dependent types. The most important idea is that something of type `D1 5` can only have 5 elements. A vector of 4 or 6 elements will not type-check.

So now that we know about shapes, let's examine layers. Layers describe relationships between our shapes. They encapsulate the transformations that happen on our data. The following are all valid layer types:

```
Relu
FullyConnected 10 20
Convolution 1 10 5 5 1 1
```

The layer `Relu` describes a layer that takes in data of any kind of shape and outputs the same shape. In between, it applies the `relu` activation function to the input data. Since it doesn't change the shape, it doesn't need any parameters.

A `FullyConnected` layer represents the canonical layer of a neural network. It has two parameters, one for the number of input neurons and one for the number of output neurons. In this case, the layer will take 10 inputs and produce 20 outputs.

A `Convolution` layer represents a 2D convolution for our neural network. This particular example has 1 input feature, 10 output features, uses a 5x5 patch size, and a 1x1 patch offset.

DESCRIBING A NETWORK

Now that we have a basic grasp on shapes and layers, we can see how they fit together to create a full network. A network type has two type parameters. The second parameter is a list of the shapes that our data takes at any given point throughout the network. The first parameter is a list of the layers representing the transformations on the data. So let's say we wanted to describe a very simple network. It will take 4 inputs and produce 10 outputs using a fully connected layer. Then it will perform an `Relu` activation. This network looks like this:

```

type SimpleNetwork = Network
  '[FullyConnected 4 10, Relu]
  '[' 'D1 4, 'D1 10, 'D1 10]

```

The apostrophes in front of the lists and D1 terms indicated that these are promoted constructors. So they are types instead of terms. To “read” this type, we start with the first data format. We go to each successive data format by applying the transformation layer. So for instance we start with a 4-vector, and transform it into a 10-vector with a fully-connected layer. Then we transform that 10-vector into another 10-vector by applying relu. That's all there is to it! We could apply another FullyConnected layer onto this that will have 3 outputs like so:

```

type SimpleNetwork = Network
  '[FullyConnected 4 10, Relu, FullyConnected 10 3]
  '[' 'D1 4, 'D1 10, 'D1 10, `D1 3]

```

Let's look at the MNIST digit recognition problem to see a more complicated example. We'll start with a 28x28 image of data. Then we'll perform the convolution layer I mentioned above. This gives us a 3-dimensional tensor of size 24x24x10. Then we can perform 2x2 max pooling on this, resulting in a 12x12x10 tensor. Finally, we can apply an Relu layer, which keeps it at the same size:

```

type MNISTStart = MNISTStart
  '[Convolution 1 10 5 5 1 1, Pooling 2 2 2 2, Relu]
  '[D2 28 28, D3 24 24 10, D3 12 12 10, D3 12 12 10]

```

Here's what a full MNIST example might look like (per the README on the library's Github page):

```

type MNIST = Network
  '[ Convolution 1 10 5 5 1 1, Pooling 2 2 2 2, Relu
    , Convolution 10 16 5 5 1 1, Pooling 2 2 2 2, FlattenLayer, Relu
    , FullyConnected 256 80, Logit, FullyConnected 80 10, Logit]
  '[' 'D2 28 28, 'D3 24 24 10, 'D3 12 12 10, 'D3 12 12 10
    , 'D3 8 8 16, 'D3 4 4 16, 'D1 256, 'D1 256
    , 'D1 80, 'D1 80, 'D1 10, 'D1 10]

```

This is a much simpler and more concise description of our network than we can get in Tensor Flow! Let's examine the ways in which the library uses dependent types to its advantage.

THE MAGIC OF DEPENDENT TYPES

Describing our network as a type seems like a strange idea if you've never used dependent types before. But it gives us a couple great perks!

The first major win we get is that it is very easy to generate the starting values of our network. Since it has a specific type, we can let type inference guide us! We don't need any term level code that is specific to the shape of our network. All we need to do is attach the type signature and call `randomNetwork`!

```
randomSimple :: MonadRandom m => m SimpleNetwork
randomSimple = randomNetwork
```

This will give us all the initial values we need, so we can get going!

The second (and more important) win is that we can't build an invalid network! Suppose we try to take our simple network and somehow format it incorrectly. For instance, we could say that instead of the input shape being of size 4, it's of size 7:

```
type SimpleNetwork = Network
  '[FullyConnected 4 10, Relu, FullyConnected 10 3]
  '[ 'D1 7, 'D1 10, 'D1 10, `D1 3]
-- ^^ Notice this 7
```

This will result in a compile error, since there is a mismatch between the layers. The first layer expects an input of 4, but the first data format is of length 7!

```
Could not deduce (Layer (FullyConnected 4 10) ('D1 7) ('D1 10))
  arising from a use of 'randomNetwork'
from the context: MonadRandom m
  bound by the type signature for:
      randomSimple :: MonadRandom m => m SimpleNetwork
  at src/IrisGrenade.hs:29:1-48
```

In other words, it notices that the chain from D1 7 to D1 10 using a FullyConnected 4 10 layer is invalid. So it doesn't let us make this network. The same thing would happen if we made the layers themselves invalid. For instance, we could make the output and input of the two fully-connected layers not match up:

```
-- We changed the second to take 20 as the number of input elements.
type SimpleNetwork = Network
  '[FullyConnected 4 10, Relu, FullyConnected 20 3]
  '[ 'D1 4, 'D1 10, 'D1 20, 'D1 3]

...

{- /Users/jamesbowen/HTensor/src/IrisGrenade.hs:30:16: error:
  • Could not deduce (Layer (FullyConnected 20 3) ('D1 10) ('D1 3))
    arising from a use of 'randomNetwork'
  from the context: MonadRandom m
    bound by the type signature for:
        randomSimple :: MonadRandom m => m SimpleNetwork
    at src/IrisGrenade.hs:29:1-48
-}
```

So Grenade makes our program much safer by providing compile time guarantees about our network's validity. Runtime errors due to dimensionality are impossible!

TRAINING THE NETWORK ON IRIS

Now let's do a quick run-through of how we actually train this neural network. We'll use the Iris data set. We'll use the following steps:

Write the network type and generate a random network from it
Read our input data into a format that Grenade uses
Write a function to run a training iteration. Run it!

1. WRITE THE NETWORK TYPE AND GENERATE NETWORK

So we've already done this first step for the most part. We'll adjust the names a little bit though. Note that I'll include the imports list as an appendix to the post. Also, the code is on the grenade branch of my Haskell Tensor Flow repository in `IrisGrenade.hs`!

```
type IrisNetwork = Network
  '[FullyConnected 4 10, Relu, FullyConnected 10 3]
  '[ 'D1 4, 'D1 10, 'D1 10, 'D1 3]

randomIris :: MonadRandom m => m IrisNetwork
randomIris = randomNetwork

runIris :: FilePath -> FilePath -> IO ()
runIris trainingFile testingFile = do
  initialNetwork <- randomIris
  ...
```

2. TAKE IN OUR INPUT DATA

The `readIrisFromFile` function will take care of getting our data into a vector format. Then we'll make a dependent type called `IrisRow`, which uses the `S` type. This `S` type is a container for a shape. We want our input data to use `D1 4` for the 4 input features. Then our output data should use `D1 3` for the three possible categories.

```
-- Dependent type on the dimensions of the row
type IrisRow = (S ('D1 4), S ('D1 3))
```

If we have malformed data, the types will not match up, so we'll need to return a `Maybe` to ensure this succeeds. Note that we normalize the data by dividing by 8. This puts all the data between 0 and 1 and makes for better training results. Here's how we parse the data:

```

parseRecord :: IrisRecord -> Maybe IrisRow
parseRecord record = case (input, output) of
  (Just i, Just o) -> Just (i, o)
  _ -> Nothing
where
  input = fromStorable $ VS.fromList $ float2Double <$>
    [ field1 record / 8.0, field2 record / 8.0, field3 record / 8.0, field4 record / 8.0]
  output = oneHot (fromIntegral $ label record)

```

Then we incorporate these into our main function:

```

runIris :: FilePath -> FilePath -> IO ()
runIris trainingFile testingFile = do
  initialNetwork <- randomIris
  trainingRecords <- readIrisFromFile trainingFile
  testRecords <- readIrisFromFile testingFile

  let trainingData = mapMaybe parseRecord (V.toList trainingRecords)
  let testData = mapMaybe parseRecord (V.toList testRecords)

  -- Catch if any were parsed as Nothing
  if length trainingData /= length trainingRecords || length testData /= length testRecords
  then putStrLn "Hmmm there were some problems parsing the data"
  else ...

```

3. WRITE A FUNCTION TO TRAIN THE INPUT DATA

This is a multi-step process. First we'll establish our learning parameters. We'll also write a function that will allow us to call the train function on a particular row element:

```

learningParams :: LearningParameters
learningParams = LearningParameters 0.01 0.9 0.0005

-- Train the network!
trainRow :: LearningParameters -> IrisNetwork -> IrisRow -> IrisNetwork

```



```
trainRow lp network (input, output) = train lp network input output
```

Next we'll write two more helper functions that will help us test our results. The first will take the network and a test row. It will transform it into the predicted output and the actual output of the network. The second function will take these outputs and reverse the oneHot process to get the label out (0, 1, or 2).

```
-- Takes a test row, returns predicted output and actual output from the network.
testRow :: IrisNetwork -> IrisRow -> (S ('D1 3), S ('D1 3))
testRow net (rowInput, predictedOutput) = (predictedOutput, runNet net rowInput)

-- Goes from probability output vector to label
getLabels :: (S ('D1 3), S ('D1 3)) -> (Int, Int)
getLabels (S1D predictedLabel, S1D actualOutput) =
  (maxIndex (extract predictedLabel), maxIndex (extract actualOutput))
```

Finally we'll write a function that will take our training data, test data, the network, and an iteration number. It will return the newly trained network, and log some results about how we're doing. We'll first take only a sample of our training data and adjust our parameters so that learning gets slower. Then we'll train the network by folding over the sampled data.

```
run :: [IrisRow] -> [IrisRow] -> IrisNetwork -> Int -> IO IrisNetwork
run trainData testData network iterationNum = do
  sampledRecords <- V.toList <$> chooseRandomRecords (V.fromList trainData)
  -- Slowly drop the learning rate
  let revisedParams = learningParams
    { learningRate = learningRate learningParams * 0.99 ^ iterationNum }
  let newNetwork = foldl' (trainRow revisedParams) network sampledRecords
  ....
```

Then we'll wrap up the function by looking at our test data, and seeing how much we got right!

```
run :: [IrisRow] -> [IrisRow] -> IrisNetwork -> Int -> IO IrisNetwork
run trainData testData network iterationNum = do
  sampledRecords <- V.toList <$> chooseRandomRecords (V.fromList trainData)
  -- Slowly drop the learning rate
  let revisedParams = learningParams
    { learningRate = learningRate learningParams * 0.99 ^ iterationNum }
```

```
let newNetwork = foldl' (trainRow revisedParams) network sampledRecords
let labelVectors = fmap (testRow newNetwork) testData
let labelValues = fmap getLabels labelVectors
let total = length labelValues
let correctEntries = length $ filter ((==) <$> fst <*> snd) labelValues
putStrLn $ "Iteration: " ++ show iterationNum
putStrLn $ show correctEntries ++ " correct out of: " ++ show total
return newNetwork
```

4. RUN IT!

We'll call this now from our main function, iterating 100 times, and we're done!

```
runIris :: FilePath -> FilePath -> IO ()
runIris trainingFile testingFile = do
  ...
  if length trainingData /= length trainingRecords || length testData /= length testRecords
  then putStrLn "Hmmm there were some problems parsing the data"
  else foldM_ (run trainingData testData) initialNetwork [1..100]
```

COMPARING TO TENSOR FLOW

So now that we've looked at a different library, we can consider how it stacks up against Tensor Flow. So first, the advantages. Grenade's main advantage is that it provides dependent type facilities. This means it is more difficult to write incorrect programs. The basic networks you build are guaranteed to have the correct dimensionality. Additionally, it does not use a “placeholders” system, so you can avoid those kinds of errors too. This means you're likely to have fewer runtime bugs using Grenade.

Concision is another major strong point. The training code got a bit involved when translating our data into Grenade's format. But it's no more complicated than Tensor Flow. When it comes down to

the exact definition of the network itself, we do this in only a few lines with Grenade. It's complicated to understand what those lines mean if you are new to dependent types. But after seeing a few simple examples you should be able to follow the general pattern.

Of course, none of this means that Tensor Flow is without its advantages. Tensor Flow has much better logging utilities. The Tensor Board application will then give you excellent visualizations of this data. It is somewhat more difficult to get intermediate log results with Grenade. There is not too much transparency (that I have found at least) into the inner values of the network. The network types are composable though. So it is possible to get intermediate steps of your operation. But if you break your network into different types and stitch them together, you will remove some of the concision of the network.

Also, Tensor Flow also has a much richer ecosystem of machine learning tools to access. Grenade is still limited to a subset of the most common machine learning layers, like convolution and max pooling. Tensor Flow's API allows approaches like support vector machines and linear models. So Tensor Flow offers you more options.

CONCLUSION

Grenade provides some truly awesome facilities for building a concise neural network. A Grenade program can demonstrate at compile time that the network is well formed. It also allows an incredibly concise way to define what layers your neural network has. It doesn't have the Google level support that Tensor Flow does. So it lacks many cool features like logging and visualizations. But it is quite a neat library for its scope.

This concludes our series on Haskell and machine learning! If you want to get started writing some code yourself, the best place to start would be our Haskell Tensor Flow Guide. It will walk you through a lot of the tricks and gotchas when first getting Tensor Flow to work on your system. You can also take a look at the Github repository and examine all the different code examples we used in this series.

APPENDIX: COMPILER EXTENSIONS AND IMPORTS

```
{-# LANGUAGE DataKinds #-}  
{-# LANGUAGE BangPatterns #-}  
{-# LANGUAGE TupleSections #-}  
{-# LANGUAGE GADTs #-}  
  
import      Control.Monad (foldM_  
import      Control.Monad.Random (MonadRandom)  
import      Control.Monad.IO.Class (liftIO)  
import      Data.Foldable (foldl'  
import      Data.Maybe (mapMaybe)  
import qualified Data.Vector.Storable as VS  
import qualified Data.Vector as V  
import      GHC.Float (float2Double)  
import      Grenade  
import      Grenade.Core.LearningParameters (LearningParameters(..))  
import      Grenade.Core.Shape (fromStorable)  
import      Grenade.Utils.OneHot (oneHot)  
import      Numeric.LinearAlgebra (maxIndex)  
import      Numeric.LinearAlgebra.Static (extract)  
  
import      Processing (IrisRecord(..), readIrisFromFile, chooseRandomRecords)
```