

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

# 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)
```

---

Revision #1

Created 11 March 2022 06:30:21 by gasick

Updated 11 March 2022 17:11:17 by gasick