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

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)
```

Revision #1

Created 2022-03-11 06:33:43 UTC by gasick

Updated 2022-03-11 17:11:17 UTC by gasick