Tensor Flock: A Short Description of this Language

February 2, 2018

1 Introduction

TensorFlock is an attempt to marry numerical and functional programming. To date, numerical programming has been dominated by DSLs like Matlab and Julia, or libraries written in other dynamically typed, procedural languages, like Python's numpy. Despite the obvious connection between numerical programming and actual mathematical functions, efforts to do linear algebra in purely functional languages tend to go awry. Haskell's hmatrix library, for example, exports five separate operators: $< . >, \# >, < \#, !\# >$, and $<>$, for matrix and vector operations. Good luck figuring out what they do without looking them up in documentation.

The main reason that Haskell defines such an awkward interface for numerical programming is its type system. Strong, static types do not allow for easy operator definitions between arbitrary objects. This is not a problem with dynamically typed languages, but the tradeoff comes in the form of runtime errors, some of which can occur several hours, or even days, into a computation, forcing the whole program to be restarted.

Given these challenges, TensorFlock seeks to demonstrate that there a exists a better path, by giving programmers a more intuitive interface for manipulating tensors within the realm of functional programming. Primarily, we intend to show this by treating tensors as primitives. By implementing a dependent type system, Tensorflock type checks tensors of arbitrary rank at compile time, and thereby hopefully saves programmers from the maddeningly frustrating experience of long computations crashing. Tensorflock also intends to make it easier for programmers to concisely represent operations with tensors in code by making use of the Einstien summation convention. The primary unit of computation, is unsurprisingly a Tensor[1]. For the purposes of this language, a Tensor can be thought of as an object with rank r that maps an r-tuple of natural numbers to a real, or complex number. Rank 1 tensors correspond to vectors and rank 2 tensors correspond to matrices. Using this concept of rank, we can have arbitrary higher dimensional tensors of any rank, or even rank 0 tensors, which are simply scalars.

[1]For the mathematically inclined, the technical definition of a tensor is, unhelpfully, "something that transforms like a tensor." This language takes the simpler treatment of a map from indices to scalars and avoids dealing with the concepts of contravariance or covariance found in General Relativity, or other fields that make heavy use of tensors.

2 Features

2.1 Einstein Summation Convention

We make use of the Einstein summation convention to allow us to concisely express all linear algebra operations with extreme concision. The convention is to sum over all repeating indices in a single term of an expression. For example, to express the squared norm of a vector $x \in \mathbb{R}^n$ using this convention, one would write

$$|\mathbf{x}|^2 = \sum_{i=1}^{n} x_i^2 = x_i x_i$$

Matrix products are

$$\mathbf{Ax} = A_{ij} x_j$$

and the dot product is

$$\mathbf{v} \cdot \mathbf{u} = v_i u_i$$

2.2 Arbitrary Component Selection

When doing tensor operations, we benefit from the ability to refer to both a single component of a tensor and the entire entity simultaneously. T_{ijk}, for example, refers to the i'th row, j'th column, and k'th "depth index[2]" of a rank three tensor, but since i, j, and k are left unspecified, T_{ijk} is functionally equivalent to the entire object. This allows us to define something akin to numpy's arange(int n), which returns an array of ints from $[0, n)$, simply as

$$A_i = i$$

If we want the numbers to be [1,n], then we simply have

$$A_i = i + 1$$

(Evidently, tensors in Tensorflock are 0-based)

2.3 Dependent Types

Usually, when we program with tensors, their size and shape are not known at compile time, but can only be determined at runtime. This leads to shape mismatches and out of bounds indexing that can crash a program, or even worse, not crash the program and simply read from uninitialized locations in memory. The only feasible way to catch such errors at compile time is to treat the types of tensors as first class values, and have the compiler warn us that we cannot, for example, have a dot product of two rank 1 tensors whose shapes are n and m, respectively, as the operation is not well defined unless $m = n$, which is not true in general.

[2]No one seems to agree on what to call this dimension https://www.gamedev.net/forums/topic/581995-silly-terminology-musing-a-3d-grid-has-rows-columns-and-/

3 Syntax and Features

TensorFlock's syntax is extremely similar to Idris', which is in turn closely related to Haskell's.

3.1 Primatives

3.2 Datatypes

The only primitive types in TensorFlock are Ints, Doubles, Chars, Strings, and Tensors. The decision to make Tensors primitives is somewhat strange, as they are complex entities that also contain rank and shape information. However, it is also important to the efficiency of our applications that Tensors be laid out contiguously in memory, and TensorFlock does not provide any way for a programmer to manage memory. Every other type can be defined by the user, or in the standard library. The syntax for such type definitions is similar to Haskell's Generalized Algebraic DataTypes (GADTs) or Agda's data declarations. Booleans, for example, can be defined like so:

```
data Bool : Type where
     False: Bool
     True : Bool
```

This declares a Boolean which has the type of Type (which is a necessary construct in a dependently typed language) with two constructors, False, and True. For a more interesting datatype, we can define the natural numbers:

```
data Nat : Type where
    Zero : Nat
    Succ : Nat -> Nat
```

a homogeneous list of polymorphic type List a:

```
data List :  Type -> Type where
    Nil : List a
    _::_ : a -> List a -> List a
```

as well as the vector type, which in this context simply means "List of fixed length:"

```
data Vect : Nat -> Type -> Type where
    Nil : Vect Zero a
    _::_ : a -> Vect k a -> Vect (Succ k) a
```

These definitions showcase a very interesting feature of the language where pattern matching can occur even in types, as demonstrated by applying the Successor function to the type variable "k" in the (::) constructor of a vector. Note that we also intend to support overloading, as demonstrated here in the case of (::) working as the Cons function for both lists and vectors.

3.3 Functions

Top level functions must have their type specified with a colon (:).

```
max : Int -> Int -> Int
max x y = if x > y then x else y
```

TensorFlock also steals Agda's "mixfix" syntax to specify operators. The if-then-else construct, for example, can be defined

```
if_then_else : Bool -> a -> a
if_then_else True expr1 _ = expr1
if_then_else False _ expr2 = expr2
```

Now, we can use if_then_else as we would expect to use it in Haskell,

```
if cond then expr1 else expr2
```

Arguments to "mixfix" functions can be given to the function after the name of the function, or in place of the underscores. The above example also illustrates the pattern matching and underscore wildcard features of the language.

TensorFlock also features let bindings, which are useful for naming intermediate computations performed in a function. For example:

```
gcd : Int -> Int -> Int
gcd x 0 = x
gcd x y = let x' = x % y in gcd y x'
```

3.4 Classes

How we actually go about supporting overloading is an interesting question. Idris simply allows for ad hoc overloading, whereas Haskell requires that the operators be defined in a "class," which loosely corresponds with the more familiar Interface from Java. It remains to be seen, then, if we implement overloading by creating the construct

```
class Consable (t : Type -> Type) where
        _::_ : a -> t a -> t a
```

3.5 Syntactic Sugar

In order to avoid having to write out lists as

```
myList : List Int
myList = 1 :: 2 :: 3 :: Nil
```

we provide the syntax

```
myList : List Int
myList = [1,2,3]
```

In order for tensor selection to be fully type safe, the general type of a tensor has to be

```
data Tensor : (Num a) => a -> (rank : Nat) ->
              (Vect rank (Nat -> Type)) -> Type
```

This complicated object a fully generalized tensor, parameterized over the numerical type of the contents, the rank, and the shape. The second line of this definition constrains the shape, which is a list of length "rank," holding functions that map from the naturals to the type of finite sets, defined as

```
data Fin : Nat -> Type where
     FZ : Fin (Succ k)
     FS : Fin k -> Fin (Succ k)
```

The type Fin n captures the types of all natural numbers in the range $[0, n)$, and is perfect for building a type safe tensor indexing function, as trying to access an out of bounds index is a type error, and the program that tries to perform this operation won't compile. The reason that the tensor type holds a vector of *functions* from Nats to Types as opposed to concrete types is that tensor dimensions do not need to be homogeneous, and each dimension of the tensor can have the type of a different member of the finite set family.

Obviously, writing out this tensor type in all of its gory detail every time one wants to use a tensor is undesirable, especially in a language constructed specifically to manipulate these objects. Thus, we provide the much more convenient syntax

```
myTensor : Tensor Int <n>
```

to denote the 1-rank tensor of size n. To declare a tensor of arbitrary rank, yet still be able to pattern match on its shape, we provide

```
generalTensor : (Num a) => Tensor a {r = Succ k} <d_1..d_r>
```

The two dots in the angle brackets are a range comprehension, defining the size of dimensions 1 through r, and the {r = Succ k} term constrains the rank of the tensor to be at least 1, since <d_1..d_0> would make no sense, and 0 rank tensors are isomorphic to scalars, which have no shape. We additionally use the ".." sugar to create list ranges, such as

```
[1..5] == [1,2,3,4,5]
```

4 Sample code

We present the building blocks of a linear algebra library using our tensors.

4.1 Utility Functions

```
-- The functor class defines things that can be mapped over
-- i.e. map (*2) [1,2,3] == [2,4,6]
class Functor (f : Type -> Type) where
    map : (m : a -> b) -> f a -> f b

instance Functor List where
        map f [] = []
        map f (x :: xs) = f x :: map f xs
```

```
instance Functor (Vect n) where
        map f Nil = Nil
        map f (x :: xs) = f x :: map f xs

-- list concatenation
_++_ : [a] -> [a] -> [a]
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

replicate : Nat -> [a] -> [[a]]
replicate Zero _ = []
replicate (Succ k) lst = lst :: (replicate k lst)

-- (n - k) is of Type Nat, which guarantees that n >= k
drop : (k : Nat) -> Vect n a -> Vect (n - k) a
drop Zero v = v
drop (Succ k) (x :: xs) = drop k xs

zipWith : (a -> b -> c) -> Vec n a -> Vec n b  -> Vec n c
zipWith f Nil Nil = Nil
zipWith f (x :: xs) (y :: ys) = (f x y) :: (zipWith f xs ys)

foldl1 : (a -> a -> a) -> Vect {r = Succ k} a -> a
foldl1 f (x :: Nil) = x
foldl1 f (x :: xs) = f x  (foldl1 f xs)

-- This tails function does not behave exactly like the one in
-- Haskell's Data.List. In fact, it is only useful as a building
-- block in the flattening function.
tails : Vect {r = Succ (Succ k)} a -> Vect (Succ k) [a]
tails vec = zipWith drop [1..(r-1)] (replicate (r-1) vec)

product : Vect {r = Succ k} <d_1..d_r> -> Nat
product <d_1..d_r> =
        -- unwrap the dimensions from their Fin constructors
        let dims = map (\ Fin x -> x) <d_1..d_r>
        in foldl1 (*) dims
```

```
-- Now, we can flatten a tensor.
-- What we would like to do, is given a tensor T<d_1..d_r>, return
-- T'<i> = T< i// prod [d_2..d_r], i// prod[d3..d_r], .., i//d_r, i % d_r>
-- where // denotes integer division and % is the modulo operator
flatten : (Num a) => Tensor a {r = Succ (Succ k)} <d_1..d_r>
        -> Tensor a 1 (product <d_1..d_r>)
flatten T<d_1..d_r> =
        let sizes = map product (tails <d_1..d_r>) in
        let indices = map (i //) (sizes) ++ [i % d_r] in
        let T'<i> = T<indices> in
        T'<i>
```