A System of Shapes Makes For Simpler Array Programming

Xuanyi Chew
chewxy@gmail.com

July 2, 2021

Abstract

ABSTRACT HERE

That a system of shapes simplifies array programming.

1 Introduction

Gorgonia is a family of libraries which brings the ability to create and manipulate deep neural networks to the Go programming language. This paper concerns two libraries in the family: `gorgonia.org/gorgonia` and `gorgonia.org/tensor`. The former is a library to define abstract mathematical expressions while the latter provides multidimensional array programming capabilities.

Recently both libaries were augmented with an algebra of shapes, which provides constraints to the array programming operations, leading to a more correct implementation of neural networks. This paper describes said algebra.

2 Multidimensional Arrays, Their Shapes and Their Fundamental Operations

Multidimensional arrays may be described by their shape.

Example. For example, a matrix $\mathbf{A}$ can be described by the number of rows r and the number of columns c. A shorthand notation would be (r, c).

In the example above, (r, c) is the **shape** of the matrix $\mathbf{A}$. We say the shape of $\mathbf{A}$ has two **dimensions**, or that it is **rank**-2. When addressing r in the shape, we'll also call it **axis** 0, while c is considered **axis** 1.

The usual, single dimensional array is a special case of a multidimensional array. By way of analogy, one may interrogate the fundamental operations of multidimensional arrays. Table 1 enumerates the analogies of fundamental operations between multidimensional arrays and unidimensional arrays. Operations after the thick line indicates utility functions that are best treated as fundamental operations.

Name of Operation	Multidimensional Array	Unidimensional Array
Size Descriptor	Shape	Length
How many elements to skip to the next index	Strides	1
Rank/Dimensions	D	1
Indexing	Takes a coordinate of size D	Index with one number
Slicing	Takes D ranges	Takes one range
Transposition	Applies a permutation of D axes	Only one permutation possible
Concatenation	Concatenation along a given axis	`append` at the end of the array

Table 1: Analogies of operations

2.1 Size Descriptor

The length of a unidimensional array denotes the size of the array. For multidimensional arrays, the length needs to be specified for each dimension. This is called a **shape**. The `.Shape()` method returns the shape of a multidimensional array. Built in slices are interpreted as unidimensional arrays, with the length acting as its shape. All other values, with the exception of maps are assumed to be scalar shaped.

2.2 Ranks/Dimensions of

A unidimensional array has only one dimension, hence the array is of rank-1, or it has 1 dimension. D is a function that takes an array, unidimensional or multidimensional, and returns the number of dimensions. In code, it is written as a method, `.Dims()`.

2.3 Indexing

Indexing allows for access to a particular element of an array. In code, it is written `A[idx]` or `A.Index(idx)`, where `A` is the multidimensional array, and `idx` is a list of numbers denoting the indices. It is important to note that indexing is different from slicing. Indexing returns an element in the array, while slicing returns another array.

In a unidimensional array, indexing simply takes a single integer. Thus `A[1]` returns the first element of the array `A`. In a multidimensional array, the indices that are required are exactly the dimensions of the multidimensional array.

We can generalize the notion of indexing with the following function signature:

$$Index : Array\ a \to [int] \to a$$

The function signature - though inadequate to describe the operation in full - nonetheless tells us that it is a function that takes a multidimensional array

with elements of type a, a list of integers, and returns a single element of type a.

As an example, consider the following matrix of integers:

$$\mathbf{A} := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Assuming a 0-based indexing system, let's say we want the 1st element of the 1st row. The function call to perform indexing is as follows: `A.Index(1,1)`. This will result in `5`, which is the (1,1)-th element of the matrix $\mathbf{A}$. By analogy, the (0,1)-th element of $\mathbf{A}$ is `2`.

The shape signature of the indexing operation is as follows:

$$Index : a \to idx \to (),\ s.t.\ (D\ idx\ = D\ a\ \wedge \forall\ (idx < a))$$

Read this as: $Index$ is a function that takes a shape (denoted by the variable a), and a list of indices (denoted by the variable idx), and returns (). This function is subject to the constraints that the dimensions of a and dimensions of idx are the same and that all values of idx is smaller than the values of a.

2.4 Slicing

$$Slice : Array\ a \to [range] \to Array\ a$$

2.5 Transposition

Unidimensional arrays do not support the transposition operation. Thus there are no analogues for transposition. This is the first novel operation that can only occur in higher dimensions. This subsection briefly analyzes the operation.

We begin with a two dimensional array, commonly known as a matrix. Let us use this matrix for example:

$$\mathbf{A} := \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

The transposition of the matrix $\mathbf{A}$ is defined as reflecting the values of the matrix along its central diagonal, so that

$$\mathbf{A}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Thus, if a matrix is of shape (a, b), the transposition would yield (b, a) as a resulting shape.

When involving multidimensional arrays where there are more than 2 dimensions, transposition requires additional parameters.

3 The Algebra of Shapes

The Shape Algebra is described by a BNF:

$$
\begin{aligned}
E &::= a \mid S \mid E \to E \mid (E\ s.t.\ P) \mid F \mid U \\
S &::= () \mid (Sz,) \mid (Sz,\ S) \mid (S,\ Sz) \mid A \\
A &::= (a,) \mid (a,\ A) \mid (A,\ a) \mid (B,) \mid (B,\ A) \mid (A,\ B) \\
B &::= E\ O_A\ E \mid \Sigma\ E \mid \Pi\ E \mid \\
O_A &::= + \mid \times \mid - \mid \div \\
F &::= I\ G\ E \mid Sl\ G\ E \mid D\ E \\
U &::= T\ Axs\ E \mid R\ Ax\ n\ E \mid Cat\ Ax\ E\ E \\
G &::= Sz \mid Sz : Sz \mid Sz : Sz : Sz \mid G, G \\
P &::= C_L\ O_c\ C_R \\
C_L &::= E \mid Axs \mid D\ Axs \mid D\ E \mid D\ G \mid \Sigma\ E \mid \Pi\ E \mid C_L\ O_c\ C_L \\
C_R &::= C_L \mid Sz \mid Axs \mid n \\
O_c &::= \wedge \mid \vee \mid = \mid \neq \mid < \mid \leq \mid > \mid \geq \\
Axs &::= Ax \mid Axs, Axs \\
Sz, Ax, n &::= \mathbb{N}
\end{aligned}
$$

4 Unification

$$\frac{a \notin E}{a \sim E : \{a/E\}}\ (1)$$

$$\frac{}{a \sim a : \{\}}\ (2)$$

$$\frac{E_1 = E_2}{E_1 \sim E_2 : \{\}}\ (3)$$

Unification 4 4 4 represents ...

5 Inference

We expect all functions to be well annotated, so inference for expresssions are less important.

Only variables and application really matters:

$$\frac{x : E \in \Gamma}{\Gamma \vdash x : E}\ \text{(Var)}$$

$$\frac{\Gamma \vdash f := E_1 \to E_2 \qquad \Gamma \vdash x : E_1}{\Gamma \vdash f@x : E_2}\ \text{(App)}$$

Mnemonic	Name
E	**E**xpression
S	**S**hape
A	**A**bstract Shape
B	
O_A	**A**rithmetic **O**peration
F	**F**undamental operation
U	**U**tility operations
I	**I**ndex of
T	**T**ranspose of
Sl	**Sl**ice of
G	Slice Ran**G**e
R	**R**epeat
Cat	Con**cat**enate
Ax	**Ax**is
P	**P**redicate
C	**C**onstraints (left and right)
O_C	**C**omparison **O**peration (used in Subject-to Clauses)
D	**D**imensions of
Sz	**Siz**e

Table 2: Mnemonics used in the BNF

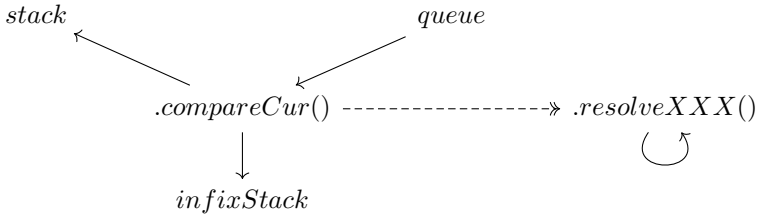
6 Semantics

7 Constraint Solving

8 Parsing

The shape languaage is designed for a fairly straightforwards parsing algorithm. The shapes package comes with a parser. The code in its entirety can be found in `parser.go`. A brief description follows.

The parser works by using two stacks (one for values and one for operators) and a queue. The folowing flowchart presents the big-picture ideas:



The main entry point to parsing is the `Parse()` function. In it, a `parser` object is created. The parsing process starts by lexing - turning a string into a slice of tokens, which forms the queue. Then an item is dequeued off the queue

and is placed onto either the `stack` or `infixStack`. The rules of when and how each token is turned into a value is described by the methods of the `parser` object.

The main workhorse is the `.compareCur()` method, which compares the operator precedence of tokens of the current value off the queue to the top of the `infixStack`. The parser also has two main groups of methods, denoted by the prefix of the methods. The `.resolveXXX` methods generally use the existing values and operators on the `stack` and `infixStack` to create new values that are then pushed into the `stack`. The `.expectXXX` methods also does lookaheads in the queue and manage the queue pointers in order to create values and use them immediately. .compareCur() calls these `.resolveXXX` and `expectXXX` methods in order to create values to put onto the `stack`.