MPL: Matrix Processing Language

David Rincon-Cruz(Language Guru),
Chi Zhang(Tester),
Jiangfeng Wang(Team Leader),
Wode Ni(System Architect)
{dr2884, cz2440, jw3107, wn2155}@columbia.edu

February 9, 2017

1 Introduction

In this project, we are going to implement a programming language, MPL, designated to matrix manipulation.

Matrices are widely used in various applications including physics, statistics, computational graphics etc. Inspired by MATLAB and mathematica, we want to allow easy and efficient matrix operations and image processing via matrix manipulation. The coding style is a combination of C language and MATLAB, and uses an imperative style with strong static typing.

The main goal of this language is to implement and achieve a simple and ease-of-use programming language for matrices computation and image processing. The programs we are targeting with this language are similar to MATLAB and Mathematica. We want to make it simple for people to solve problems which involve working extensively with matrices. For example, the language is extremely easy and helpful to implement a linear equation problem solver or matrices calculation.

In addition to solving matrices problems, the language also provides image processing feature. In this case, each image is treated as an array of pixels, and self-defined functions can be applied to each pixel of the image. Furthermore, we allow a matrix of different functions to be applied on corresponding pixels of the image, allowing regions of a single image to be processed differently.

2 Language Features

The fundamental data type in MPL is matrix, and MPL allows simple while customizable matrix operations. In traditional general-purpose languages, these operations often involve an extensive use of nested loop structures and conditional statements. In MPL, we introduce the idea of having functions as entries in the matrix and design simple syntax to populate, manipulate, and apply those matrices. MPL treats entry functions as first class objects, so we can populate a matrix by functions and directly apply this function matrix to ordinary matrices with the same dimensionality. The language will run an implicit loop over every entry of the matrix and evaluate the corresponding entry function, which eases the user from writing nested for loops.

In our current design, to reduce unnecessary complexity, the language is imperative and there

is no C-like functions. All user-defined functions operate only on entries of matrices.

In MPL, an entry function has the access to the entry it currently operates on and nearest 8 neighbors. We design an `@` operator that applies a matrix of functions to an ordinary function. Since we frequently want to apply the same operations on all entries, MPL provides `.@` operator that applies one function to all entries.

The information about the neighbors enable MPL to perform operations such as convolution, which is frequently used in image processing. To properly refer to the current entry and its neighbors in an entry function, MPL provides a series of keywords starting with `#`. The current entry can be accessed by `#C`, and the neighbors are `#N`, `#W`, `#E`, `#S`, `#NW`, `#NE`, `#SW`, `#SE`. The `#` prefix is to resemble the configuration of the nine entries.

3 Syntax

3.1 Comments

The comment style will be `C` and `Java` Fashion with single line and block-style comments.

Table 1: Comment Styles in MPL

Type	Name
`// COMMENT`	Comments out rest of current line
`/* COMMENT */`	Comments out section in between delimiters. Is nestable.

3.2 Types

There are 3 "primitive" types that are the building components for our higher structures.

Table 2: Table of Types Available for MPL

Type	Name
`int`	Integer
`float`	Floating Point Number
`boolean`	Boolean

`int` and `float` will be implemented in their 32-bit representations, while `boolean` will be stored in tradition 1-bit form.

3.3 Special Types

A key aspect of the mpl language is "equivalency" of function and primitive types.

Functions, have strict typing but can be overloaded to take Integer argument and Float arguments.

Functions are matrix-oriented and hence take implicit arguments as they're meant to apply to entries.

In order to access the argument the function is applied on, the syntax is `#C`. Likewise, as the element is defined on a matrix entry with neighbors, the neighbors can likewise be factored into the computations with similar access codes.

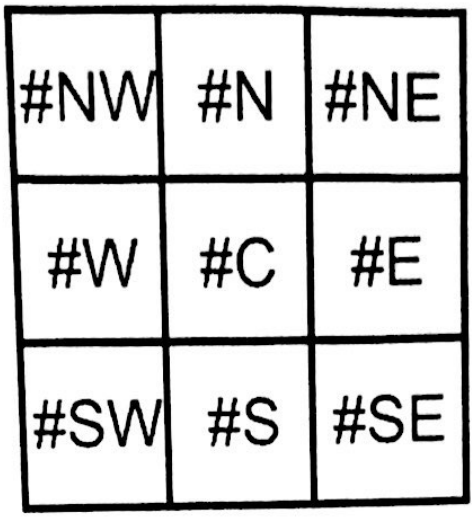

Figure 1: Given a function applied to the center, these are the access codes for each entry.

All functions must have a single return value (which is strictly typed). Functions must be declared with an explicit return type (which is how overloading separates same-name functions) and will have all their code delimited with braces. An example of having every entry average its value with that of it's neighbors:

```
int func fAverage {
  int average = (#NE+#N+#NW+#W+#C+#E+#SW+#S+SW)/9;
  return average;
}
```

Awk-like

3.4 Structures

There are two fundamental structure that MPL is built around: Matrix(`Mat`), Images(`Img`).

3.4.1 Mat

Matrices are the high level equivalent of their math counterparts and will be singly typed as one of 3: integer matrices, float matrices, and function matrices. Function matrices can be applied to integer and float matrices, and standard matrix operations apply (operators defined subsequently).

3.4.2 Img

Images are the next higher level structure to `Mat`. Images are a collection of "channel" matrices: R, G, B, A. Operations applied to Images are applied to each channel individually or each channel can be reference individually: `Img1.R`.

3.5 Operators

MPL consists of 9 key operators:

Table 3: Table of Operators Available for MPL

Operator	Name
+	Addition
-	Subraction
/	Diision
*	Multiplication
'	Transpose
./	Element-by-Element Division
.*	Element-by-Element Multiplication
@	Application
.@	Element-by-Element Application

The difference between @ and @. is in regards to arguments. @. takes a function matrix and a value matrix and applies functions to corresponding entries in the value matrix. @ takes a single function and applies it to every entry of a value matrix.

4 Sample Code

4.1 Image blur sample

In this example, we construct a function that performs convolution on an image using a blur kernel encoded.

```
Img image = imgread("Lena.png");

func fblur {
  double grayValue=0.0;
  grayValue = #C*0.147761 + (#N+#S+#E+#W)*0.118318 + (#NE+#NW+#SE+#SW)
      *0.0947416
    return grayValue;
}

Mat blurredR = fblur @ R;
Mat blurredG = fblur @ G;
Mat blurredB = fblur @ B;

Img output = new Img(blurredR, blurredG, blurredB);

imgwrite(output,"blurredLena.png");
```

4.2 Matrix of functions sample

In this section, we demonstrate the semantics of matrix functions and related operations on them.

```
func f1 {
  int temp = #C + 1;
    return temp;
}

func f2 {
  int temp = #C + 5;
    return temp;
}
func f3 {
  int temp = #C - 7;
    return temp;
}
func f4 {
  return temp;
}

Mat A = { 1 2;
      3 4;}
Mat B = { 1 2;
      3 4 }
Mat C = {1 1;
      1 1;}

Mat F = { f1 f2;
      f3 f4;}
```

– specify regions? or use other operation, like – color edge blue, background red, specify regions using same

└ What's the difference w/ @.

```
Mat D = F @ C;

print D;

Mat E = A * F @ C;

print E;

Mat F1 = {f1 f2;
       f3 f4};
Mat F2 = {f1 f2;
       f3 f4};
```

```
Mat Famazing = F1 * F2;

Mat result = Famazing @ C;
```