Macaw

Mathematical Calculation Language

William Hom wh2307
Yi Jian yj2376
Joseph Baker jib2126

Introduction and Motivation

Many commercial languages support basic mathematical types like integers, floats, and vectors (as arrays typically). However, popular languages often require additional libraries to allow users to perform matrix operations, even if the language supports multi-dimensional arrays. Often only specialized mathematical languages like Matlab or R have full support for matrix calculations. Macaw is a Matlab- and R-inspired language that provides a matrix data type and by allowing a simple operator overloading mechanism is able to provide a standard library of matrix manipulation functions.

Language Description

Macaw is a strongly typed, imperative language that supports simple flow control, looping, and user defined function constructs. User-defined functions can be aliased to operators to allow vector and matrix computation to resemble the equations that users are familiar with.

Language Keywords

`Float, Int, Matrix, String, Vector, else, for, function, if, in, print, raise, return, while`

Data Types / Structures:

```
# Denotes Comments

# Integers
a <- 5;
b <- 9;

# Floats
f <- 8.0;
```

```
# Vectors
v <- [1, 2, 3];
w <- 1:4;
# w = [1, 2, 3, 4]
x <- [](3);
# x = [0, 0, 0]
# Strings
s <- “Hello ”;
t <- "There";
```

```
# Matrices
m <- [1, 3, 6; 8, 2, 0];
# m = [1, 3, 6;
#      8, 2, 0]
n <- [;](2,5);
# n = [0, 0, 0, 0, 0;
#      0, 0, 0, 0, 0]
```

Operators

```
# Comparison: a and b must be
# same type. Returns 1 or 0
# Language supports for Int and
# Float
a = b;
a != b;
a < b;
a <= b;
a > b;
a >= b;

# 'Boolean' Composition: non-0
# Int/Float are true, 0 = false
# Returns 1 or 0
a & b;
a | b;
```

```
# Math: a and b must be same type
# Language supports these for
# Int and Float
a + b;
a - b;
a * b;
a / b;

# String Concat
s + t;

# Parens
(a + b) + c;
```

Array/Matrix Access

```
# Access indices are 1-based
a <- v[2];
# a = 2
b <- m[2;2];
# b = 2

# Assignment
v[2] <- 9;
# v = [1, 9, 3]
m[1;1] <- 3;
# m = [3, 3, 6;
#      8, 2, 0]
```

```
# Shape
length <- v.len;
row_count <- m.rowlen;
col_count <- m.collen;
```

Exceptions

```
# raise halts execution with the
# given error message
raise "Can't divide by zero!";
```

Printing

```
# Print outputs argument to the
console
a <- 5;
print a;
# a = 5
```

Flow Control

```
# If / Else
if a = b {
  a <- 5;
}
else {
  a <- 8;
}
```

```
# Loops
sum <- 0;
for i in 1:v.len {
  sum <- sum + v[i];
}
while a != 0 {
  a <- a - 1;
}
```

Functions

```
# User defined function
function gcd(Int a, Int b) {
  while a != b {
    if a > b {
      a <- a - b;
    }
    else {
      b <- b - a;
    }
  }
  return a;
}

a <- gcd(12, 8);
# a = 4
```

```
# Recursive user defined function
function factorial(Int n) {
  if n == 0 {
    return 1;
  }
  else {
    return factorial(n - 1) * n;
  }
}

b <- factorial(4);
# b = 24
```

Operator Extending

```
# function declaration followed
# by the ~ symbol and the
# operator to extend. See note
# below for more info.
function matrix_add(Matrix a,
Matrix b) ~ + {
  c = [;](a.rowlen, a.collen);
  for i in 1:a.rowlen {
    for j in 1:a.collen {
      c[i;j] <- a[i;j] + b[i;j];
    }
  }
  return c;
}

a <- [1,2;3,4];
b <- [5,6;7,8];
c <- a + b;
# c = [6, 8;
#      10, 12]
```

```
function matrix_transpose(Matrix
a) ~ ' {
  b <- [;](a.collen, a.rowlen)
  for i in 1:a.rowlen {
    for j in 1:a.collen {
      b[j;i] <- a[i;j];
    }
  }
  return b;
}

a <- [1,2;3,4;5,6]
# a = [1,2;
#      3,4;
#      5,6]
b <- a';
# b = [1,3,5;
#      2,4,6]
```

Note about Operator Extension

Operator extension simply allows a user-defined function to be aliased by a symbolic operator and allows for a nicer rearrangement of the arguments. Rather than calling `matrix_add(a, b)`, users can invoke that function by calling `a + b`. The types of `a` and `b` determine which function will be used. When invoking the function, if the function is a unary function, the alias follows the single argument. If the function has a pair of arguments, the alias symbol is in between the arguments. Only certain operators are valid for function extending via the ~ operator and language operators cannot be overridden (for instance a user cannot alias a function onto `Int + Int`). Extensible operators are (in order of decreasing precedence):

```
' ^ * .* / ./ % + - = != > < >= <=
```

Operator Associativity

Composition of an operators is left-associative. For example, `a + b + c` is evaluated in the same way as `(a + b) + c`.

Sample Program

We intend to write a standard library with common matrix operations similar to the addition and transpose function we wrote above in the operator extension section. The following program assumes that transpose, dot product, and addition function have been defined for arguments of matrix, vector, and integer types.

The following program uses Taylor's Theorem to approximate the values of a function `sin(x) + sin(y)` around the coordinate `(0.5, 0.5)`. Using this theorem we are able to approximate this function without having access to the sin function. We simply have to precompute some matrices which we have included. This shows that with even the basic matrix operations users can write code to approximate a rich set of mathematical functions.

```
# v is a vector of size 2 [x, y]
# approximate_sin returns an approximation of sin(x) + sin(y) at
# (0.5,0.5) + v using Taylor's Theorem
function approximate_sin(Vector v) {
     H <- [-0.4794, 0; 0, -0.4794];
     g <- [0.8776, 0.8776];
     f <- 0.9589;

  approx <- f + (g * v') + 0.5 * ((H * v')' * v');
  # approx is a 1x1 matrix
  return approx[1;1]
}

a <- approximate_sin([0.02, 0.05]);
# a = 1.01964[1]

b <- approximate_sin([0.1, 0.09]);
# b = 1.1213
```

[1] Answer calculated using Wolfram Alpha with the expected behavior of our language and algorithms with this user defined function.