DATUM

K S Ananth
CS22BTECH11029
IIT Hyderabad

Nimai Parsa
CS22BTECH11044
IIT Hyderabad

Kartikeya Mandapati
CS22BTECH11032
IIT Hyderabad

M Bhavesh Chowdary
CS22BTECH11041
IIT Hyderabad

Arugonda Srikar
CS22BTECH11008
IIT Hyderabad

Gunnam Sri Satya Koushik
CS22BTECH11026
IIT Hyderabad

Mohammed Gufran Ali
CS22BTECH11040
IIT Hyderabad

September 2024

Supervisor **Dr. Ramakrishna Upadrasta**

Abstract

This whitepaper presents a domain-specific language(DSL) which is designed for tasks related to data manipulation and visualization. With the recent developments in machine learning and data science applications, the importance of data is ever growing and is needed for every model, may it be a large model made by professionals or a small model made by a beginner. Which is why there is a need for a language that is intuitive, brief, and user friendly so that the user can preprocess the data as required before using the same. Therefore our DSL provides these and more by offering features such as intuitive and chainable syntax, strong semantic analysis, and pipeline functions, enabling users to create readable, and less verbose data transformation processes. Possible optimizations of the DSL are lazy evaluation, vectorization, memory-efficient data slicing, and the ability to easily integrate various data transformation functions. The language focuses on making the code intuitive and readable by adding simple English-like syntax. All of the above discussed aspects about the DSL is explained in detail in this whitepaper.

Acknowledgements

We would like to express our gratitude as a team to our course instructor, **Dr. Ramakrishna Upadrasta**. For the guidance and support throughout this project and course. We also extend our thanks to our Teaching Assistants(TA's) **Mr. Rajapu Jayachandra Naidu** and **Mr. Rajiv Shailesh Chitale** for their insights on the DSL and always being available for any doubts. Finally, we acknowledge the course CS3423 Compilers-II at the Indian Institute of Technology, Hyderabad for providing the opportunity to do this project.

Contents

Chapter 1

Introduction

1.1 What is DATUM:

DATUM is a simple and complete domain specific language which circles around data manipulation, visualization, pre-processing and data handling while focusing on ease of use and intuitiveness.

1.2 Motivation

As ML and DS continue to evolve, so does the demand for the languages and tools. But many existing languages and tools are often complex, less productive, bulky or lack some processing to do some specific data related tasks.
name is developed to address some of these issues by providing productive, less complex language with possible optimizations and it also provides data visualization.

1.3 Ease of Use:

Programming in name is simple. The language provides a special data type 'dataset' and features that can be used on them.

1.4 Completeness:

The DSL is complete i.e. the user can program their own functions and manipulate the data accordingly.

1.5 Intuitive:

The language provides a new operator '→' which eases the use of language as it suggests the flow of data through features enabling pipelining which helps in understanding the code better and also helps in debugging. It also offers common English keywords more on this is covered later.

Chapter 2

Aim of our DSL

There already exist several libraries for data manipulation and visualization. The motivation of creating a DSL when there already exists libraries is covered in the following points:

- Availability of all features at one place. The language offers some features that may not be present in one library, but present in other libraries.
- It is not function oriented rather it is language oriented where the user can come up with their own functions.
- Possible optimization offerings are Lazy Evaluation, Vectorization, Memory-efficient Data Slicing, Pipeline optimization and Reordering, which are explained later.
- The lack of strong semantic analysis is fulfilled by our DSL.
- The syntax is designed in such a way that improves the productivity of a programmer by increasing the readability and making it less verbose.

The way we achieve this is explained in the following sections.

2.1 Making Intuitive Syntax

The syntax and keywords in our DSL are designed to be intuitive and self-explanatory, making it easy for users to understand and follow. For instance,

```
1 row(from 1 to 10)
```

clearly expresses the intention to select rows 1 to 10, using plain English that is accessible even to those with limited programming experience. By prioritizing clear, human-readable commands, our DSL allows users to focus on their data rather than decoding complex syntax.

2.2 Write Concise Codes with Chainable Syntax

The chainable syntax of our DSL offers a highly intuitive and user-friendly approach to data manipulation, preprocessing, and visualization. Our DSL allows a user to chain multiple pipeline functions together, thus allowing the user to perform data manipulations / transformations in a pipelined fashion.

Example:

```
ds->row((from 1 to 5) also (from 10 to 15 step 3))
  ->col(to "W" step 2)
  ->filter({elem == 10;})
  = 100;
```

Our DSL is less verbose and complex than the traditional data manipulation frameworks. This syntax minimizes nested function calls, making data operations feel more natural and readable. Unlike existing languages, which often rely on verbose function nesting, our DSL's clear and concise chaining mechanism makes it easy for users to combine operations without losing track of their data transformations, enhancing productivity and reducing error.

2.3 Pipeline Functions

Apart from the basic pipeline functionalities provided by our DSL, we also provide the freedom to the user to create their custom pipeline functions to use them in their custom pipeline to match their specific needs.

Example:

```
function (dataset ds) -> fill_null_values(int num) -> dataset {
  ds->filter(ds == null) = num;
  return ds;
}

ds->fill_null_values(0)->rows(from 1 to 10)->show();
```

This pipeline function takes a dataset, replaces all the null values in that dataset with the user provided number and returns the dataset.

A pipeline function is declared by using the function keyword. Each pipeline function expects the datatypes of the input of the pipe, datatypes of the output of the pipe, and the list of datatypes of the additional parameters.

2.4 Strong Semantic Analysis

Our DSL offers a very good semantic analysis ensuring that the logic of the code and the manipulation of the dataset being performed by the user is correct and also makes sense. We perform semantic checks on various aspects such as data type check while performing operations, pipeline order, behavior of functions, scope and data tracking at each step.
A few other data manipulation libraries already present out there mostly use Python as their primary language which does not offer an extremely impressive semantic analysis as it gives a lot of runtime errors. In our DSL we try to perform the semantic check and reduce the same.

Some key semantic checks:

1. **Data Type check:** Ensures that all the data types in the DSL are operated only on the data types that they are allowed to operate on. For example, arithmetic operations are only allowed on numerical types such as integer, float and dataset with all data types of type integers and floats, while concatenation is limited to string. By adding strict data types rules we can limit errors such as concatenation of number with a string which happens in languages like python or JavaScript to name a few.

2. **Pipeline Ordering:** The pipeline feature of our DSL allows to chain data transformations in a logical sequence. The semantic analysis checks that each step in the pipeline operates on valid input and produces valid output for the next function in the chain.

3. **Behaviour of Functions:** This semantic check ensures that these functions behaves accordingly with their declared input and output types. This check prevents issues where functions may return incorrect types, leading to pipeline failures.

4. **Data Tracking:** As data moves through different stages of manipulation, the DSL tracks its transformations to ensure consistency. It ensures that the shape and structure remain valid for the upcoming operations.

Chapter 3

Program Structure

```
$$ Program that drops rows with null values in the first column, and fills all
    the null values in the second column with zeros. Then normalises each column
     of the dataset.
function_declarations:
function (dataset ds) -> fill_null_values(int num) -> dataset {
  ds->filter(ds == null) = num;
  return ds;
}
function (dataset ds) -> normalize() -> dataset {
    array(integer) mean_arr = ds->mean(1);
    loop i from 0 to ds->shape[0] {
        ds->col(i) -= mean_arr[i];
    }
    array(integer) std_arr = ds*ds->sum(1) / ds->shape[0];
    loop i from 0 to ds->shape[0] {
        ds->col(i) /= std_arr[i];
    }
    return ds;
}
start:
dataset ds = read("./dataset", "csv")
ds->col(1)
  ->fill_null_values(0)
  ->row({
        return r[0] == null;
    })->drop();

ds->normalize()->write("./dataset.csv", "csv");
```

The above code is a sample code using which we will explain the program structure of the DSL.

3.1 Function Declarations

All the function declarations in the code are found under the `function_declarations:` label and above the `start:` . The `function_declarations:` label is optional. We can have programs without any user defined functions.

3.2 Start of Program

The program starts executing from the line below `start:` label. The `start:` label is mandatory.

3.3 Syntax of Functions

3.3.1 Pipeline Functions

A pipeline function is a function that can be part of a pipeline.

A pipeline function is declared by using the function keyword. Each pipeline function expects the datatypes of the input of the pipe, datatypes of the output of the pipe, and the list of datatypes of the additional parameters.

```
function (datatype1 param1, ...) -> function_name(datatype1 param1, ...) -> (
    datatype1, ...) {
  $$ Statement 1
  $$ ...
  $$ Statement N
}
```

3.3.2 Parameter Functions

A parameter function is a function that can be passed as a parameter to a pipeline function. It is supported for the built-in pipeline functions like `row()`;

```
... -> function_name({
        $$ Statement 1
        $$ ...
        $$ Statement N
```

```
    }) -> ...
```

The body of the function is surrounded by '{}'. We don't have to explicitly mention the parameters of the function. The undeclared variables in the body become the function parameters.

Example:

```
(ds1, ds2)->join({
                    integer sum1 = r1->sum();
                    integer sum2 = r2->sum();
                    return sum1 != sum2;
                })
                ->row(to 10)->show();

$$ the parameter function here is equivalent to

(r1, r2) {
  integer sum1 = r1->sum();
  integer sum2 = r2->sum();
  return sum1 != sum2;
}
```

The order of the undeclared variables determine the order of the parameters of the function. The datatype of `r1` and `r2` are assumed to be arrays.

Chapter 4

Lexemes

YaLM follows some of the standard data types already present in C/C++, But it also offers some new cool data types and keywords which are domain specific and makes it easier for the programmer to use and understand.

4.1 Data types:

Datatype	Keyword	Description	Size
Integer	integer	It is the datatype to store signed numeric data values that fit within the 64-bit range.	8 Bytes (64 bits)
Float	float	The float data type is to store floating point values that fit within 64-bit representations.	8 Bytes (64 bits)
Char	char	The char type is used to store a single character using a single byte.	1 Byte (8 bits)
String	string	The string type is used to store a character string or a sequence of characters at contiguous memory locations.	Dynamic

Continued on next page

Datatype	Keyword	Description	Size
Boolean	bool	The boolean data type, which represents two possible values: true or false, used to indicate binary states or logical conditions.	1 Byte (8 bits)
Dataset	dataset	The dataset datatype is designed to store data read from CSV or TSV files, for further transformations and manipulations through pipelining and various built-in functions. It is a complex data structure used to store the entire dataset into a variable.	Dynamic
Array	array	The array data type is a dynamic container data structure designed to store a sequence of elements, where each element can be of only one single data type, including integer, float, char, string, boolean, and dataset. These elements are stored in contiguous memory locations, allowing for efficient access and manipulation.	Dynamic

4.2 Comments:

4.2.1 Single Line comments:

A single line comment starts with '$$'.

$$ This is a single line comment.

4.2.2 Multi Line comments:

Multi line comments start with '$!' and terminate with '!$'.

$! This is a comment.

This comment illustrates multi line comments. !$

4.3 Operators:

All basic numeric, relational and logical operators are supported. The following are the supported operators:

4.3.1 Arithmetic:

Operator	**Description**	**Supported Data Types**
Increment (++)	Increments the variable by the value 1.	integer, float
Decrement (–)	Decrements the variable by the value 1.	integer, float
Division (/)	Standard division	integer, float
Multiply (*)	Standard multiplication	integer, float
Modulus (%)	Modulus operator	integer
Addition (+)	Standard addition for integers and floats, offers concatenation for strings	integer, float, string (concatenation)
Subtraction (-)	Standard subtraction for integers and floats, also used to get the integer ASCII value difference between two characters or strings.	integer, float (subtraction), char, string (difference in ASCII value)
Assignment (=)	Standard assignment operator	All data types
Multiply and Assign (*=)	Multiply with a new value and assign the result to the same variable	integer, float
Divide and Assign (/=)	Divide with a new value and assign the result to the same variable	integer, float
Modulus and Assign (%=)	Perform modulo operation with a new value and assign the result to the same variable	integer
Add and Assign (+=)	Add with a new value and assign the result to the same variable	integer, float
Subtract and Assign (-=)	Subtract with a new value and assign the result to the same variable	integer, float

4.3.2 Relational Operators:

Operator	Description	Supported Data Types
Greater than (>)	Returns true when the left value is greater than the right value.	integer, float, char, string
Lesser than (<)	Returns true when the left value is lesser than the right value.	integer, float, char, string
Greater than or equal to (>=)	Returns true when the left value is greater than or equal to the right value.	integer, float, char, string
Lesser than or equal to (<=)	Returns true when the left value is lesser than or equal to the right value.	integer, float, char, string
Equality (==)	Returns true when the left value is equal to the right value.	integer, float, char, string
Inequality (!=)	Returns true when the left value is not equal to the right value.	integer, float, char, string

4.3.3 Logical Operators:

Operator	Description	Supported Data Types
Logical and (and)	Takes the logical AND of two statements. Returns true if both statements are true.	integer, float
Logical or (or)	Takes the logical OR of two statements. Returns true if at least one statement is true.	integer, float

4.3.4 Additional Keywords:

Keyword	Description
step	Used with from to syntax to skip the values in chunks of step value
also	Union of two arrays
from	The initial value

Continued on next page

Keyword	Description
to	The end value, end value is exclusive
function	Used to declare a function

4.3.5 Built-in Pipeline Functions:

input_type → Keyword → output_type	Expected Parameters
ds → row() → ds	1. Function returning boolean (parameter - array or array, int) 2. From, to, step, also syntax 3. Array of int (indices) or boolean (bitmask) 4. Single row
ds → col() → ds	1. Function returning boolean (parameter - array or array, int) 2. From, to, step, also syntax 3. Array of int (indices), string (col names), boolean (bitmask) 4. Single column
ds → filter() → ds	Function returning boolean (parameter - int or int, int)
ds → sum → array **ds → sum → int** **array → sum → int**	1. Parameter (int - axis when input is dataset) 2. No parameter when input is array
ds → max → array **ds → max → int** **array → max → int**	1. Parameter (int - axis when input is dataset) 2. No parameter when input is array

Continued on next page

input_type → Keyword → output_type	**Expected Parameters**
ds → min → array **ds → min → int** **array → min → int**	1. Parameter (int - axis when input is dataset) 2. No parameter when input is array
ds → mean → array **ds → mean → int** **array → mean → int**	1. Parameter (int - axis when input is ds) 2. No parameter when input is array
(ds, ds) → join → ds	Function returning boolean (parameters are two arrays, rows)
void → read → ds	Read ds from file
ds → write → void	Write ds to file
ds → unique → ds	Unique rows, no parameters
ds → split → array(ds)	Parameter - array of proportions
array → sort → array **ds → sort → ds**	1. int or string (to sort according to the values of a column) 2. No parameters
array → shuffle → array **ds → shuffle → ds**	Shuffle the elements. No parameters
ds → add(. . .) → ds	The parameters in the add feature must be provided in a comma separated values pattern with the right data type as that of the of the respective column
ds → shape → array	Returns [rows, cols]
ds → drop()	1. Parameter - 0 or 1, 0 is for row and 1 is for column 2. Returns void

4.3.6 Visualization:

Keyword/Function	Description	Usage
ds → col(i) → show_bar() **ds → col(i) → show_bar(array)**	1. Displays a barchart of frequencies plotted on the input column. 2. The displayed values can be constrained to the ones present in the input array.	Supported for discrete values with limited cardinality.
(ds → col(i), ds → col(j)) → show_scatter() **ds → col(i) → show_scatter(array)**	1. Creates a scatter plot to visualize the relationship between two columns. 2. The range of the values can be constrained to [l,r].	Supported for numerical data.
(ds → col(i), ds → col(j)) → show_line() **ds → col(i) → show_line([l1, r1], [l2, r2])**	1. Displays a line chart between two columns so visualize the relationship. 2. The range of the values can be constrained to [l,r].	Supported for numerical data.
(ds → col(i), ds → col(j)) → show_box() **ds → col(i) → show_box()**	1. Displays box plots to visualize the distribution of data across different categories. 2. The range of the values can be constrained to [l,r].	Supported between numerical and categorical variable type attributes.

4.4 Control Flow:

4.4.1 Conditional Flow:

if:

Executes the code block if the specified condition evaluates to true. Usage:

```
if ( condition ){
    Statement1;
    Statement2;
}
```

else if:

Used after an if condition to specify a new condition if the previous if or another else if condition present above was false. It proceeds to execute the code block if this new condition is satisfied. Usage:

```
if ( condition1 ){
    Statement1;
}
...(cascaded else if conditions)...
else if ( condition2 ){
    Statement2;
}
```

else:

Provides a default code block that executes if none of the preceding if or else if conditions are true.

```
...( if followed by (else if)* conditions )...
else {
    Statement3;
}
```

4.4.2 Iterative flow:

loop:

Iterates over a sequence or range, executing a specified block of code for each element in the sequence or iteration. Usage:

```
integer temp = 0;
loop (i from 1 to 10 step 3){
    temp++;
}
$$ temp value is 3 after the loop
```

break:

Exits from the nearest enclosing loop, immediately terminating the iteration and continuing execution with the code that follows the loop. Usage:

```
integer temp = 0;
loop (i from 5 to 10 step 4){
    temp++;
    if( condition ) break;
}
```

continue:

Skips the rest of the current iteration of a loop and proceeds with the next iteration. Usage:

```
integer temp = 0;
loop (i from 1 to 100 step 19){
    temp++;
    if( condition ) continue;
}
```

return:

Returns the data type as defined in the function. Usage:

```
function (dataset ds) -> fill_null_values(int num) -> dataset {
  ds->filter(ds == null) = num;
  return ds;
}
```

Chapter 5

Possible Optimizations

This chapter summarises the main outcomes and conclusions resulting from this body of work.

5.1 Lazy Evaluation

- Instead of executing operations immediately, we can collect all the commands in a pipeline and evaluate them only when the final result is needed. (Example: When we have to write the dataset back to the disk)
- In this way, we can improve the performance by minimizing unnecessary computations, especially when chained operations may override each other.
- In the Example given, before the second unique instruction is executed, the rows of `ds` are already unique. So the performance of this code can be optimized by the skipping redundant computation.

```
dataset ds = read("./dataset.csv", "csv");
ds = ds->unique();
ds = ds->row(to 10)->col(to 10)->unique();
ds->write("./dataset.csv", "csv");
```

5.2 Vectorization

- When performing operations to a dataset, like adding two columns or adding a scalar to a dataset, we can apply operations to entire columns or rows at once instead of iterating element-by-element.

- In this way we can speed our program by leveraging the power of multi-processing / multi-threading or GPU programming.
- In the Example given, say, the user tries to convert height from meters to centimeters. Instead of iterating over every row, we can utilize the GPU to perform a single operation on multiple data.

```
dataset ds = read("./dataset.csv", "csv");
ds.col('height') *= 100 $$ can be parallelized
```

5.3 Memory-efficient Data Slicing

- When new datasets are declared to store a result of a data transformation, we can use views or references instead of copying data.
- Copying large dataset are both time and memory intensive. Therefore this optimization reduces memory overhead and improves slicing performance.
- In the above example, three other datasets are declared. If copies were created instead, the program would take way more time and space.

```
dataset ds = read("./dataset.csv", "csv");
dataset ds1 = ds->col(to 10)
dataset ds2 = ds->col(from 10 to 20)
dataset ds3 = ds->col(from 20 to 30)
```

5.4 Pipeline Optimization and Reordering

- We can analyze the pipeline and automatically reorder operations in a pipeline for an efficient performance.
- In this way, we can increase the performance by minimizes the amount of data processed in later stages of the pipeline.
- In the example given above, if filter is applied first to the dataset, then it has to be applied to all rows and columns of the dataset. But on reordering the operations in the pipeline, we can avoid the expensive computation.

```
dataset ds = read("./dataset.csv", "csv");

ds->filter({
```

```
4           return elem = null;
5       })
6       ->row(from 5 to 10) = 0;
7
8   $$ can be reordered to
9
10  ds->row(from 5 to 10)
11    ->filter({
12          return elem = null;
13      }) = 0;
```

Chapter 6

Sample Codes

6.1 Sample Code 1:

```
$$ Program that drops all the rows whose sum is not 1 and replaces the values in the quality column with 1 if it is greater than or equal to 7 and 0 if it is less than 7

function_declarations:

function (dataset ds) -> convert_values_to_binary(int num, string colName) -> dataset {
    ds->col(colName)->filter({e < num}) = 0;
    ds->col(colName)->filter({e >= num}) = 1;
}

start:

dataset ds = read("./dataset.csv", "csv")

ds->convert_values_to_binary(7, "quality")
  ->row({
    return r.sum() != 1;
  })
  ->drop(0);

ds->show_bar();
ds->write("./dataset-updated.csv", "csv");
```

6.2 Sample Code 2:

```
$$ Program that drops rows with null values in the first column, and fills all
    the null values in the second column with zeros. Then normalizes each column
     of the dataset.

function_declarations:

function (dataset ds) -> fill_null_values(int num) -> dataset {
  ds->filter(ds == null) = num;
  return ds;
}

function (dataset ds) -> normalize() -> dataset {
    array(int) mean_arr = ds->mean(1);
    loop i from 0 to ds->shape[0] {
        ds->col(i) -= mean_arr[i];
    }
    array(int) std_arr = ds*ds->sum(1) / ds->shape[0];
    loop i from 0 to ds->shape[0] {
        ds->col(i) /= std_arr[i];
    }
    return ds;
}

start:

dataset ds = read("./dataset", "csv")
ds->col(1)
  ->fill_null_values(0)
  ->row({
        return r[0] == null;
    })->drop(0);

ds->normalize()->write("./dataset.csv", "csv");
```

6.3 Sample Code 3:

```
$$ First convert each row with their respective probabilities row-wise and split
    to train and test sets with ratios 0.8 and 0.2 respectively.

function_declarations:

function (dataset ds1) -> train_test_split(float testSize) -> (dataset,dataset)
{
    ds->shuffle();
    float trainSize = 1 - testSize;
    array(ds) splitDs = ds->split([trainSize, testSize]);
    return splitDs[0], splitDs[1];
}

function (dataset ds) -> convert_rows_to_orobabilities() -> (dataset)
{
    ds->row({r = r/r->sum()});
    return ds;
}

start:

dataset ds = read("./dataset.csv", "csv")

train, test = ds->convert_rows_to_orobabilities()->train_test_split(0.2);

X_train = train->col(to 10)
y_train = train->col(10)
X_test = test->col(to 10)
y_test = test->col(10)

X_train -> write("./x-train", "csv");
y_train -> write("./y-train", "csv");
X_test -> write("./x-test", "csv");
y_test -> write("./y-test", "csv");
```