Towards a Formal Type System for DataFrames

William W. Ma
University of Chicago
williamma@uchicago.edu

ABSTRACT

While there has been a lot of work to improve the computational efficiency of data science, little work has been done to improve the data scientists ability to process data. The tools that they use, such as pandas, has evolved from single threaded to cloud scale. However, these tools still lack a consistency in their behavior. In this paper, we introduce the beginnings of a type system for dataframes. With this type system, we will be able to formally prove systems-level optimizations, but, more importantly, provide the data scientist with a consistent type system for them to do their analysis on.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
William W. Ma. 2018. Towards a Formal Type System for DataFrames. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

With the rise of data science, dataframes rose in popularity as they were the go to data manipulation model for data analysis. In order to satisfy these data scientists, popular cloud-scale systems, such as Spark and dask, all provide dataframe interfaces. However, as dataframes have gone from single-threaded pandas dataframes to cloud-scale dataframes with SparkSQL, these dataframes all lack a consistent data model and type system. Although prior work has addressed the issue of a data model, there has been no attempt at a consistent data model [2]. While data models and type systems do not have catchy punchlines or provide 10x improvements in run time, they do provide clarity for the data scientist. Arguably, this clarity could be worth more than the 10x improvement that any system could provide.

Thus, in this paper, we introduce the beginnings of a consistent type system for dataframes. The type system on the data types is provided at both the dataframe level and individual cell level. Due to the introductory nature of this work, some simplifications have been made but this does not detract from the main idea that we can have a consistent type system for dataframes.

We also included proofs of progress and preservation for both type systems.

The rest of this paper is structured as follows. We first introduce the logical models used and simplifications made in section 2. Then, we discuss the physical implementation in Coq 8.13 in section 3. Finally, we discuss future work in section 4.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

2 LOGICAL MODEL

2.1 DataFrame Level Type System

For our purposes, we reduced a dataframe down to a tuple that consists of

- Row index, which consists of the position and label.
- Row data types (dtypes)
- Column index
- Column dtypes
- Matrix of values

Building on this data model, we reduce the possible functions on the dataframe to the following

- Transpose, which transposes the dataframe
- Mask, which gives the index positions to keep. This can be applied over the columns or the rows.
- Filter, which gives a function that returns a boolean when applied over the rows or columns. Then, only the true rows or columns are kept.
- ToLabels, which converts a given column or row index position to the corresponding axis' index.
- FromLabels, which adds a row or column that consists of the axis' index labels.
- Map, which applies a map over the rows or columns. We simplified this to also require the possible data types that each column or row could result into.
- Concat, which is the union, intersection, or difference of the columns or rows of two dataframes.
- InferDTypes, which infers the data types from the values. This can be applied to the columns or rows.

For this introudctory work, we limit the data types that a dataframe can have to be only strings and natural numbers. Thus, the data values at the dataframe level are

```
DFValue  :=
         |  DFV_String
         |  DFV_Nat
```

The terms for the type system are

```
Axis       :=
            |  Columns
            |  Rows

Index      :=  ℕ * DFValue

ConcatType :=
            |  Union
            |  Intersection
            |  Difference

DataFrame  :=
            |  DataFrame
            |  Transpose DataFrame
            |  Mask (list ℕ)
                    Axis DataFrame
            |  Filter (list DFValue → 𝔹)
                      Axis DataFrame
            |  ToLabels ℕ Axis DataFrame
            |  FromLabels Axis DataFrame
            |  Map (list DFValue → list DFValue)
                   (list DFValue) DataFrame
            |  Concat ConcatType Axis
                      DataFrame1 DataFrame2
            |  InferDTypes Axis DataFrame
```

The only value in the dataframe level type system is the dataframe constructor itself. Because every function can take in a value or step, we only list the step relation for transpose

$$\text{TransValue}\ \frac{\texttt{value df}}{\texttt{Transpose df} \longrightarrow \texttt{DataFrame (df.transpose)}}$$

$$\text{TransStep}\ \frac{\texttt{df} \longrightarrow \texttt{df'}}{\texttt{Transpose df} \longrightarrow \texttt{Transpose df'}}$$

We also include three optimizations, which are listed below:

$$\text{TransTwice}\ \frac{\texttt{value df}}{\texttt{Transpose(Transpose df)} \longrightarrow \texttt{df}}$$

$$\text{FilterNone}\ \frac{\texttt{value df}}{\texttt{Filter (fun row} \Rightarrow \texttt{true) axis df} \longrightarrow \texttt{df}}$$

$$\text{ToFromLabels}\ \frac{\texttt{value df}}{\texttt{ToLabels 0 a (FromLabels a df)} \longrightarrow \texttt{df}}$$

For the type definitions, we follow the following format for all of the terms.

$$\frac{\vdash \texttt{df} \in \texttt{row}_{\texttt{dtypes}}, \texttt{col}_{\texttt{dtypes}} \qquad \texttt{df' = Transpose df}}{\vdash \texttt{df'} \in \texttt{col}_{\texttt{dtypes}}, \texttt{row}_{\texttt{dtypes}}}$$

2.2 Data Level Type System

At the data level, we adjust the data model to use a matrix of values that is infinitely large. Thus, we have to introduce a "empty" cell value. Because we switch over to the data level, we have to define the casting rules between data types. We allow all data types to be cast into the "empty" and string data types. In addition to the above list of functions on dataframes, we also introduce a cast function for the data level type system.

Additionally, we change the semantics of some of the functions to work independently of the other data values in the row and column. For mask, filter, and map, we assume that the functions passed in can work on the individual data values independent of the other values in the row or column. It is worth noting that at the data level, Transpose, ToLabels, FromLabels, Concat, and InferDTypes are all given a new value to replace the old value with. Thus, for simplicity, we replace these functions with an Update function.

Thus, the terms, values, and types are

```
dterm   :=
         |  DEmpty dterm
         |  DString dterm
         |  DNat dterm
         |  CastDType dterm dterm dterm
         |  Mask 𝔹 dterm dterm
         |  Filter (dterm → 𝔹) dterm dterm
         |  Map (dterm → dterm) dterm dterm
         |  Update dterm dterm dterm

dvalue  :=
         |  dv_empty DEmpty
         |  dv_nat DNat
         |  dv_string DString

dtype   :=
         |  Empty
         |  String
         |  Nat
```

The step relations are as follows:

$$\text{CastEmpty}\ \frac{}{\texttt{CastDType}\ d\ \texttt{DEmpty} \longrightarrow d}$$

$$\text{CastToString}\ \frac{}{\texttt{CastDType DString}\ d \longrightarrow \texttt{DString}}$$

$$\text{CastNat}\ \frac{}{\texttt{CastDType DNat DNat} \longrightarrow \texttt{DNat}}$$

$$\text{MaskFalse}\ \frac{}{\texttt{Mask false}\ d \longrightarrow \texttt{DEmpty}}$$

$$\text{MaskTrue}\ \frac{}{\texttt{Mask true}\ d \longrightarrow d}$$

$$\text{FilterTrue}\ \frac{\texttt{CastDType}\ d_{in}\ \ d \longrightarrow d'}{\texttt{Filter}\ d_{in}\ \texttt{true}\ d' \longrightarrow d'}$$

$$\text{FilterFalse}\ \frac{\texttt{CastDType}\ d_{in}d \longrightarrow d'}{\texttt{Filter true}\ d_{in}\ \texttt{false}\ d' \longrightarrow \texttt{DEmpty}}$$

$$\text{Map}\ \frac{\texttt{CastDType}\ d_{in}\ \ d \longrightarrow d'}{\texttt{Map}\ d_{in}\ \ d_{out}\ \ d' \longrightarrow d_{out}}$$

$$\text{Update}\ \frac{}{\texttt{Update}\ d_{new}\ \ d_{old} \longrightarrow d_{new}}$$

The type definitions are listed below.

$$\text{T\_Empty}\ \frac{}{\vdash \texttt{DEmpty} \in \texttt{Empty}}$$

$$\text{T\_String}\ \frac{}{\vdash \texttt{DString} \in \texttt{String}}$$

$$\text{T\_Nat}\ \frac{}{\vdash \texttt{DNat} \in \texttt{Nat}}$$

$$\text{T\_CastEmpty}\ \frac{\vdash d \in D}{\vdash \texttt{CastDType}\ d\ \texttt{DEmpty} \in D}$$

$$\text{T\_CastToSTring}\ \frac{\vdash d \in D}{\vdash \texttt{CastDType DString}\ d \in \texttt{String}}$$

$$\text{T\_CastNat}\ \frac{}{\vdash \texttt{CastDType DNat DNat} \in \texttt{Nat}}$$

$$\text{T\_MaskFalse}\ \frac{\vdash d \in D}{\vdash \texttt{Mask false}\ d \in \texttt{Empty}}$$

$$\text{T\_MaskTrue}\ \frac{\vdash d \in D}{\vdash \texttt{Mask true}\ d \in D}$$

$$\text{T\_FilterTrue}\ \frac{\vdash d \in D \qquad \vdash d_{in} \in D_{in} \qquad \vdash d' \in D_{in} \qquad \texttt{CastDType}\ d_{in}\ d \longrightarrow d'}{\vdash \texttt{Filter}\ d_{in}\ \texttt{true}\ d' \in D_{in}}$$

$$\text{T\_FilterFalse}\ \frac{\vdash d \in D \qquad \vdash d_{in} \in D_{in} \qquad \vdash d' \in D_{in} \qquad \texttt{CastDType}\ d_{in}\ d \longrightarrow d'}{\vdash \texttt{Filter}\ d_{in}\ \texttt{false}\ d' \in \texttt{Empty}}$$

$$\text{T\_Map}\ \frac{\vdash d \in D \qquad \vdash d_{in} \in D_{in} \qquad \vdash d_{out} \in D_{out} \qquad \texttt{CastDType}\ d_{in}\ d \longrightarrow d'}{\vdash \texttt{Map}\ d_{in}\ d_{out}\ d'\ \texttt{false}\ d' \in D_{out}}$$

$$\text{T\_Update}\ \frac{\vdash d_{new} \in D}{\vdash \texttt{Update}\ d_{new}\ d_{old} \in D}$$

3 PHYSICAL IMPLEMENTATION

We implemented the type systems and dataframe operations within Coq version 8.13[1]. Proofs of progress and preservation are within their respective TypeSystem_*.v files. The implementations of the dataframe operations are within Dataframe.v. We do include an Optimizations.v file but it is empty and is there mainly for thinking about the organization of the project.

4 FUTURE WORK

For future work, we intend to extend this type system to fully represent the behavior of pandas. Given the foundation in this paper, adding additional functions in the dataframe level type system will only require implementing them within Coq. For the data level, additional operators will look a lot like the existing step relations and type definitions. For instance joins will most likely look like a filter and groupbys will end up looking a lot like maps.

This work done for this paper will lay the ground work for formal verification of dataframe optimizations. These optimizations include the fact that the transpose of an operation done on a transpose is essentially the operation with a different axis argument, assuming that the operation has an axis argument. Some of the helper theorems have been defined in Dataframe.v to demonstrate the type of theorems that will be needed to prove the optimizations that we use.

This project is also on github [1].

[1] I couldn't get auto to use a custom database so I added hints to core. I will fix this once I have time.

REFERENCES

[1] William W. Ma. 2021. dataframe-proofs. https://github.com/williamma12/dataframe_proofs.

[2] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *CoRR* abs/2001.00888 (2020). arXiv:2001.00888 http://arxiv.org/abs/2001.00888