IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 7, 2021, accepted January 21, 2022, date of publication January 25, 2022, date of current version February 9, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3146287

Enabling Type Checking on Columns in Data Frame Libraries by Abstract Interpretation

YUNGYU ZHUANG, (Member, IEEE), AND MING-YANG LU
Department of Computer Science and Information Engineering, National Central University, Taoyuan 32001, Taiwan
Corresponding author: YungYu Zhuang (yungyu@ieee.org)

This work was supported by the Ministry of Science and Technology, Taiwan, under Grant MOST 107-2221-E-008-024-MY3.

ABSTRACT Data frames are a tabular data structure widely used in transforming data to an appropriate form in data analysis, especially in data wrangling. However, when data frames are implemented with libraries rather than supported at the language level, it is hard to find out errors due to the limitation of type checking on columns. Data scientists may encounter errors due to missing column labels or inconsistent types, especially when they reuse code snippets for new data. These errors are usually left to runtime, and it is sometimes difficult to find out where the problems are. To address this issue, we propose using abstract interpretation to perform type checking on data frame columns. We defined the type for data frames based on column labels and developed semantics to verify the typing in general operations. A static checker can be implemented based on the semantics to help programmers quickly fix errors without executing the code. To show the feasibility, we implemented a proof-of-concept for the pandas library as an example, PDChecker, to discuss the limitation and usage. It is then used to compare the functionalities with existing solutions. The results show our approach can fulfill the function of type checking for data frames. Supporting more data frame operations is included in our future work.

INDEX TERMS Data frames, type checking, source code analysis, static program analysis, data wrangling, computer languages, programming, programming environments, software tools.

I. INTRODUCTION

Data wrangling, also known as data munging, is an essential step in data science. It is the process of transforming raw data to a format suitable for data analysis. In order to make data suitable and usable in the later stages of the analytic process, data have to be cleaned, extracted, merged, and reshaped. Thus, the process of data wrangling is often the most time-consuming task in data analysis, and how to effectively and efficiently transform data is a challenge actively discussed by many research [1], [2].

Data frames are a widely used data structure in data science, especially in data wrangling. It is two-dimensional and contains tabular data, including row labels and column labels. Under the hood, a data frame is a list of vectors that have the same length, and each vector represents a column of the data frame [3]. Data scientists can use data frames to store raw data, filter/merge data based on certain conditions, and universally apply specific operations to data. The idea of data frames might come from relational database management systems (RDBMS) and spreadsheet programs, but data frames are bound to programming techniques more tightly and are usually implemented with object-oriented programming. Data frame objects are first proposed in the White book of S language [4] and followed by R language [5]. As what a relational database can do, data frames support various relational operations such as union, difference, and intersection. However, the data in data frames are ordered and have no primary key. Data frames are used as other data structures in programming and can benefit from several programming techniques like method chaining and higher-order function mapping (apply-to-all). Besides R language, recently data frames are also supported in many languages and libraries for data analysis, for example the pandas library [6] on Python language, the Frames library [7] on Haskell language, the frameless library [8] on Scala language, and Gamma language [9]. Among these languages, Python seems the most often used and popular one to recommend an aspiring data scientist to learn first, according to the worldwide survey conducted by Kaggle in these years [10], [11]. The features attracting programmers might include dynamically typing, C-like syntax, and code readability [12]. Most of all, a large

The associate editor coordinating the review of this manuscript and approving it for publication was Sabah Mohammed.

number of high-quality libraries and good community support make Python suitable for programming in research and industry projects [13]. For example, SciPy Stack [14], [15], the packages of which form an ecosystem that makes it easy to carry out the whole process of data analysis, including NumPy [16], [17], Matplotlib [18], IPython [19], and pandas [6]. Programmers can conduct relational operations as in SQL [20] and seamlessly pass data between libraries for analysis and visualization; no explicit transformation is needed.

However, providing data frames with libraries such as pandas lacks the language support for checking the information in columns. The columns of data frames are the basis of manipulating data since they usually contain the same information for different subjects, which means any mismatch in data frame columns during data manipulation will result in errors. As a library implementation, especially in dynamically typed languages, such column errors can hardly be figured out in an early stage. Even though the columns can be considered as kind of types for data frames, they are treated as values and determined at runtime. Furthermore, programming in libraries like pandas is considerably error-prone due to its complex design of API [21], [22], which may lead to obscure error messages. The causes of common errors include the lack of tracing column labels and types, passing flag arguments with the string type, and the lack of type information of the functions given as arguments.

Although migrating to languages with a rigorous type system and modern abstraction features, such as Haskell and Scala, is an effective solution, it might not sound convincing to data scientists. First, these languages are usually considered a steeper learning curve. Whether learning these languages is more complex or not, data scientists hardly choose any language other than Python and R, and learning a new language is never easy. Second, data scientists often reuse code snippets to construct new programs to analyze new data. For those who collect a lot of useful code snippets, it will be tough to migrate. This observation led us to propose statically checking the columns of data frames in libraries. The contributions of this study are two-fold. First, we point out the column issues in data frame libraries and present how to figure out these errors by abstract interpretation [23], [24]. Second, we define the type for data frame objects and use it to develop semantics to check the correctness of data frame operations. We also demonstrate how a static checker can be implemented and integrated with existing development environments to help programmers.

The remainder of this paper is organized as follows. Section II shows a motivating example to point out the column errors concretely. Section III presents our approach and explains how it checks the typing in general data frame operations, along with the preliminary and the limitation. Section IV gives a proof-of-concept implementation for pandas as an example of our approach. Section V evaluates our approach by discussing the functionality of type checking for data frames and comparing it with existing work, along with a user scenario. In Section VI, several techniques toward solving the issue we address are discussed as related work. Finally, Section VII concludes this paper and mentions future directions.

II. MOTIVATING EXAMPLE

In a library supporting data frames, such as pandas, data frames are usually supported with the objects in object-oriented programming, specifically defining a data frame class, where every column in it is represented with a series class. As shown in Figure 1, a series object is a one-dimensional array with row labels (`'a'`, `'b'`, and `'c'` in this example), and a data frame object can be thought of as a dictionary-like container for series objects [25], which maps column labels (`'category'`, `'price'`, `'weight'`, and `'stock'` in this example) to series objects. When loading data into data frame objects, records are stored in rows and every column contains the same type of information for different records. Programmers can call various methods on data frame objects to manipulate data based on columns. For example, the series in Figure 1(a) can be obtained by specifying the column label `'price'` on the data frame in Figure 1(b), which contains only the integers in the column `'price'` for every record. Column labels are usually used as a keyword to characterize the information stored in the same column and even imply the type of stored values.

'a'	20
'b'	10
'c'	30

(a)

	'category'	'price'	'weight'	'stock'
'a'	'apple'	20	0.9	True
'b'	'banana'	10	0.7	False
'c'	'cherry'	30	0.5	True

(b)

FIGURE 1. An example of series (a) and data frame (b).

However, the error checking on columns of data frame objects is not sufficient due to the lack of language support. Although column labels can be regarded as some sort of type for records, in the library they are managed as values rather than types and thus any column error will be left to runtime. In other words, invalid operations such as specifying a missing column label will be a runtime error. Unlike type errors, runtime errors are more difficult to resolve since they are mixed up with logic problems. It means that programmers might suffer from errors caused by incorrect labels or types of columns, and the messages can hardly help debugging. Taking Python and pandas as an example, programmers might write the code in Figure 2 for their data wrangling tasks. This piece of code is to perform a sequence of operations starting with a data frame object named `df` using the method chaining technique:

1. drop duplicate rows in `df` based on columns labeled `'x'` and `'y'`,

2. merge with another data frame object named **df2** with a database-style join done on the columns **'x'** and **'y'** and using only keys from the left frame,
3. group by the columns **'x'** and **'y'** in SQL-style (**as_index=False**), and
4. aggregate using the operation named **'sum'** over the column labeled **'z'**.

According to the semantics of object-oriented programming and a general library design, every method call returns a data frame object, which will then be used to perform the following operation specified by the method call after the dot. This code seems good and might work well in some cases, but it has potential problems with column labels and types. An experienced programmer who knows data frames well may figure out the following potential problems:

a. Do these labels, **'x'**, **'y'**, and **'z'**, really exist in the specified data frames? (Line 1, 2, 4, and 5)
b. Is it correct to use **'left'** for the parameter **how**? (Line 3)
c. Do the two data frames really have the common columns **'x'** and **'y'**? (Line 2)
d. Is the type of function specified in the **agg** method consistent with the type of the target column, **'z'**? (Line 5)

Data scientists are usually forced to find out the causes after running such code with actual data and encountering these problems. These problems are due to the lack of checking the label and type of columns, even though most operations on data frames are based on them.

```
1 df.drop_duplicates(['x', 'y'])
2   .merge(df2, on=['x', 'y'],
3          how='left')
4   .groupby(['x', 'y'], as_index=False)
5   .agg({'z': 'sum'})
```

FIGURE 2. An example of using data frames in data wrangling tasks.

A possible approach is to revise the above library design in a statically typed language with advanced type features as follows. 1) Breaking the **merge** method into several ones named according to its parameter values or wrapping parameter values up in enumerated types to avoid specifying parameters with strings. 2) Considering the label and type of columns with dependent types to involve columns in data frame types. Then we may write the code shown in Figure 3. It looks pretty similar to the one in Figure 2, but

- the spelling error in a string such as **'left'** and **'sum'** can be found in advance since they are the method names now (the potential problems a and b),
- no runtime error on referring to the wrong columns since the type of data frames with **'x'** and **'y'** can be distinguished from the ones whose column labels are **'x'** and **'z'** (the potential problem c), and
- no runtime type error occurs when applying the specified functions since the types of the **sum** function and the column **'z'** can be retrieved (the potential problem d).

```
1 df.drop_duplicates(['x', 'y'])
2   .leftJoin(df2, on=['x', 'y'])
3   .groupby(['x', 'y'], as_index=False)
4   .agg({'z': sum})
```

FIGURE 3. It is possible to check the types of columns by the compiler in a statically typed language with dependent types.

However, this approach requires migration to a language with dependent types and might not be realistic to data scientists. Moreover, runtime errors cannot be prevented if column information in the data frames **df** and **df2** depend on file contents rather than program code.

Suppose a static type checker based on abstract interpretation can be provided to consider column information in data frame libraries as types. In that case, programmers can verify their code before executing it with extensive data. The checker may even try to open files and read the first row for the case that data frames depend on an external data source. It can then be integrated into development environments for showing hints and warnings. Since data scientists often reuse code snippets to analyze new data, such a checker can help them quickly find out the differences in data and fix the errors accordingly.

So far as we know, there is no existing research activity on such a static checker for data frame libraries. The reasons might include the following. First, data frames are rarely discussed in research activities. Unlike SQL, which was developed based on relational algebra, data frames lack standards and formal semantics despite remarkable success. Even though data frames were introduced almost three decades ago and implemented in different languages, there is no standard for them. Different implementations might have different designs and behaviors. Second, the current type system of external checkers for languages like Python is not powerful enough. Although type hints have been introduced for third-party static checkers, for example PEP 484 for Python [26], column labels cannot be regarded as types. This motivated us to develop the idea of interpreting data frame code with alternative semantics for checking column labels and types.

III. A STATIC TYPE CHECKER FOR DATA FRAMES

To find out column errors in data frame libraries in an early stage, we propose verifying column labels and types with a static checker by abstract interpretation. For a code piece written with data frame libraries, we can use a static checker to model data frames and interpret the operations on them to perform type checking. Instead of directly building a new type system for its host language, our approach can retain compatibility with existing programs and libraries.

A. ABSTRACT INTERPRETATION

Abstract interpretation is a technique to interpret a given piece of code with different semantics, which enables the structure sharing between static analysis methods and runtime computation [23], [24]. In other words, programs written to denote runtime computation can also be used to describe

computations in another abstract universe. For example, the following line of code do simple arithmetic:

$$(\texttt{-1}) \ ^* \ \texttt{2} \ ^* \ (\texttt{-3})$$

Compilers and interpreters generally evaluate the expression to a numeric value, **6**; it is based on the semantics of the language we are using. Instead, we can define an abstract universe that has the rule of signs to interpret it as follows:

$$\texttt{(-)} \ ^* \ \texttt{(+)} \ ^* \ \texttt{(-)}$$

which results in **(+)**. It is useful when we care about only the sign of numbers but not exact numbers. Similarly, the same expression can be understood based on the semantics for type checking:

$$\texttt{int} \ ^* \ \texttt{int} \ ^* \ \texttt{int}$$

In this case, checkers examine whether the type **int** can be accepted by the operator * or not. This technique makes it possible to perform the execution, proofs, and analysis on the same code.

B. TYPE DEFINITION OF DATA FRAMES AND SERIES

Following the idea of abstract interpretation, we define a type for data frames and interpret their operation code based on the semantics of type checking. Note that the type is not the data type, i.e. class, defined in data frame libraries for data manipulation but the one used in a static checker for type checking. Below we first consider series type and define data frame type based on it since data frame objects can be regarded as a dictionary-like container for series objects.

We can define the type of a series object as the join of its elements' type since these data are the same information for different data records. In other words, all elements of a series object share the same type, which exactly represents the series object's type. Suppose a set of variables is given to construct a series object named s as follows:

$$s = \mathit{Series}(x_1, \ldots, x_\mathrm{n})$$

where the types of $x_1, \ldots, x_\mathrm{n}$ are $T_1, \ldots, T_\mathrm{n}$ in the given environment Γ, respectively. The type of the generated series object s can then be defined with a single type T_join, which is the join, i.e. the least common supertype, of $T_1, \ldots, T_\mathrm{n}$:

$$T_\mathrm{join} = T_1 \wedge \ldots \wedge T_\mathrm{n}$$

It means that any value typed with T_1 or $\ldots$or T_n can also be typed with T_join. All data in the series object can be characterized with the type T_join, and thus it is used as the type for the series object. This type checking rule is described as:

$$\frac{\Gamma \vdash x_1 : T_1, \ldots, x_n : T_n \quad \Gamma \vdash T_1 \wedge \ldots \wedge T_n = T_{join}}{\Gamma \vdash \mathit{Series}(x_1, \ldots, x_n) = s : T_{join}} \text{(T-S-Creation)}$$

After defining the type of series objects, we can then define the type of data frame objects as a set of type mapping. Suppose $l_1, \ldots, l_\mathrm{n}$ are column label strings and the types of series objects $s_1, \ldots, s_\mathrm{n}$ are $T_1, \ldots, T_\mathrm{n}$ in the environment Γ. When column labels and series objects are given in pairs to construct a data frame object as follows:

$$df = \mathit{DataFrame}(\{l_1, s_1\}, \ldots, \{l_\mathrm{n}, s_\mathrm{n}\})$$

the generated data frame object's type can be denoted by:

$$\{l_i \rightarrow T_i^{i \in 1..n}\}$$

which maps a given column label to the type of its corresponding series object. The rule for data frame creation is described as:

$$\frac{\Gamma \vdash l_1 : \mathit{String}, \ldots, l_n : \mathit{String} \quad \Gamma \vdash s_1 : T_1, \ldots, s_n : T_n}{\Gamma \vdash \mathit{DataFrame}(\{l_1, s_1\}, \ldots, \{l_n, s_n\}) = df : \{l_i \rightarrow T_i^{i \in 1..n}\}} \text{(T-Creation)}$$

Note that we use a mapping rather than a single type to represent data frames' type; it is a composite of series types. Furthermore, we treat column labels as a part of type rather than strings. Label literals are included in the data frame type and used in type checking.

C. TYPE CHECKING FOR DATA FRAMES AND SERIES

Based on the types of data frames and series, we can check the correctness of typing in their operations. For element-wise operations, we can define the following T-S-ElemOp rule to ensure that the types of operands are consistent and the operation results in a series object with the same type:

$$\frac{\Gamma \vdash s_1 : T \quad \Gamma \vdash s_2 : T}{\Gamma \vdash s_1.\mathit{elemop}(s_2) = s_3 : T} \text{T-S-ElemOp}$$

For applying a given function to a series object typed T_1, the function's type must take T_1 as the parameter type. Suppose the function is typed $T_1 \rightarrow T_2$, the resulted series object must be typed T_1 as described by the following T-S-Apply rule:

$$\frac{\Gamma \vdash s : T_1 \quad \Gamma \vdash f : T_1 \rightarrow T_2}{\Gamma \vdash s.\mathit{apply}(f) = s' : T_2} \text{T-S-Apply}$$

The type checking rules for general operations on data frames [27], including selection, projection, union, join, group-by, and apply, are listed in Figure 4. The selection operation is to select data records according to a given predicate, so the type of its result should be consistent with the one it operates on as described in T-Selection rule:

$$df.\mathit{select}(\mathit{pred}) = df' : T$$

where df is typed with T. The projection operation is to project based on the specified columns, and T-Projection describes the case of only a specific column, resulting in a series object:

$$df.\mathit{project}(x) = s : T_k^{1 \le k \le n}$$

where x is a column label string. For projection with multiple columns, we can combine T-Projection rule with T-Creation rule to perform the check. The union operation is to simply concatenate data frames with the same type, resulting in a data frame with the same type $\{l_i \rightarrow T_i \ ^{i \in 1..n}\}$ as described

in T-UNION rule. On the other hand, the join (merge) operation is to generate a data frame based on the union of the two sets of data frame column labels, A and B:

$$\{l_k \rightarrow T_k^{k \in A \cup B}\}$$

which can be described as T-JOIN rule. As to the group-by operation, it is to generate a group of data frames according to the value in a specified column. In this case, the types of these grouped data frames are the same as the original type until the next operation is performed.

$$\frac{\Gamma \vdash df : T \quad pred : Boolean}{df.select(pred) = df' : T} \text{ T-SELECTION}$$

$$\frac{\Gamma \vdash df : \{l_i \rightarrow T_i^{i \in 1..n}\} \quad \Gamma \vdash x : String = l_k^{1 \le k \le n}}{df.project(x) = s : T_k^{1 \le k \le n}} \text{ T-PROJECTION}$$

$$\frac{\Gamma \vdash df_1 : \{l_i \rightarrow T_i^{i \in 1..n}\} \quad \Gamma \vdash df_2 : \{l_i \rightarrow T_i^{i \in 1..n}\}}{\Gamma \vdash df_1 \cup df_2 = df_3 : \{l_i \rightarrow T_i^{i \in 1..n}\}} \text{ T-UNION}$$

$$\frac{\Gamma \vdash df_1 : \{l_i \rightarrow T_i^{i \in A}\} \quad \Gamma \vdash df_2 : \{l_j \rightarrow T_j^{j \in B}\}}{\Gamma \vdash df_1 \bowtie df_2 = df_3 : \{l_k \rightarrow T_k^{k \in A \cup B}\}} \text{ T-JOIN}$$

$$\frac{\Gamma \vdash df : \{l_i \rightarrow T_i^{i \in 1..n}\} \quad \Gamma \vdash x : String = l_k^{1 \le k \le n}}{\Gamma \vdash df.groupby(x) = \{df_1 : \{l_i \rightarrow T_i^{i \in 1..n}\}, ..., df_n : \{l_i \rightarrow T_i^{i \in 1..n}\}\}} \text{ T-GROUPBY}$$

$$\frac{\Gamma \vdash df : \{l_i \rightarrow T_i^{i \in 1..n}\} \quad \Gamma \vdash T_1 \wedge ... \wedge T_n = T_{join} \quad \Gamma \vdash f : T_{join} \rightarrow T_{ret}}{\Gamma \vdash df.apply(f) = df : \{l_i \rightarrow T_{ret}^{i \in 1..n}\}} \text{ T-APPLYELEM}$$

$$\frac{\Gamma \vdash df : \{l_i \rightarrow T_i^{i \in 1..n}\} \quad \Gamma \vdash f : T_1, ..., T_n \rightarrow T_{ret}}{\Gamma \vdash df.apply(f) = s : T_{ret}} \text{ T-APPLYROW}$$

$$\frac{\Gamma \vdash df : \{l_i \rightarrow T_i^{i \in 1..n}\} \quad \Gamma \vdash T_1 \wedge ... \wedge T_n = T_{join} \quad \Gamma \vdash f : T_{join} \rightarrow T_{ret}}{\Gamma \vdash df.apply(f) = s : T_{ret}} \text{ T-APPLYCOL}$$

FIGURE 4. The semantics of type checking for data frame objects.

The apply operation is to aggregate the data using functions. It can avoid explicit iteration on values of data frames and might remind users of the MapReduce programming model [28]. When applying a function to a data frame, there are three cases. The first one is applying the function to every element in the data frame, which means the function should take a join of all column types and its return type will be used to check every element's type:

$$f : T_{join} \rightarrow T_{ret}$$

where $T_{\text{join}} = T_1 \wedge \ldots \wedge T_{\text{n}}$. The result is still a data frame whose type is a mapping from any given label to the function's return type, as described in T-APPLYELEM rule:

$$df : \{l_i \rightarrow T_{ret}{}^{i \in 1..n}\}$$

The second one is applying the function to each row to reduce the information for every data record. In this case, the function has to take all column types as its parameter type:

$$f : T_1, \ldots, T_n \rightarrow T_{ret}$$

and its return type must match the type of generated series, as described in T-APPLYROW rule:

$$s : T_{ret}$$

The last one is applying to every column to aggregate the information of all data records based on each column as described in T-APPLYCOL rule. The function type in this case will be similar to the one in T-APPLYELEM rule since the function needs to accept all column types as the input. However, the result will be a series with its return type as in T-APPLYROW rule. These type checking rules may be further combined to check various operations, and extending them to verify more sophisticated operations in data frame libraries is included in our future work.

IV. PDCHECKER: A PROOF-OF-CONCEPT IMPLEMENTATION FOR PANDAS

In order to show the feasibility and usability of our approach, we implemented a proof-of-concept named PDChecker[1] for the pandas library as an example. Note that the type checking semantics for data frames may be applied to any data frame library, but here we took the case of pandas and Python to explain implementation details, discuss the limitation, and show the usage concretely. Our draft implementation also follows Language Server Protocol (LSP) to implement a frontend that can be used in most environments including JupyterLab [29].

A. TYPE-LEVEL CLASSES AND CHECK FUNCTIONS

PDChecker accepts all the code that follows the syntax of Python but evaluates them based on the type checking semantics for data frames. As shown in Figure 5, we can consider Python interpreter (a) and PDChecker (b) as functions that accept a piece of Python code as input and return something as output, following the idea in the referred papers [30], [31]. The output is a value in the case of the Python interpreter, while the output of our checker is a type. Given a piece of pandas code, PDChecker will replace the import of pandas library with its check module and interpret code with its own classes and methods for type checking.

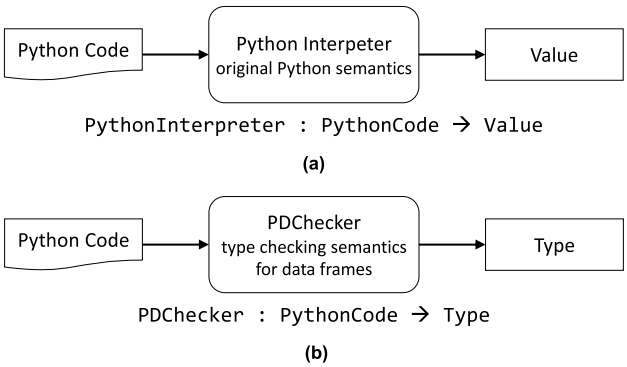


FIGURE 5. PDChecker interprets code based on the type checking semantics for data frames.

We prepared type-level classes to represent the types of data frames and series in pandas, respectively. Figure 6 shows the definition of type-level classes and parts of check functions for the explanation. As shown in Line 3, the

[1]The PDChecker Project. https://github.com/ncu-psl/pdchecker

class DataFrame has a field named **columns** for mapping between labels and their types. Note that both the type and the value, i.e., the label string, are traced in DataFrame to avoid losing label information during the type evaluation. The method calls to data frame objects will then be replaced with the ones defined in the type-level classes. Line 5-23 shows an example of how we check the column labels and types for the call to the **merge** method. It first checks the string given in the parameter **how**, gets column labels in the parameter on, and examines whether the specified labels exist in both data frames or not. Furthermore, the common columns' types must be consistent; otherwise, an error will be thrown. On the other hand, for the Series class, we only need to trace its type as shown in Line 26 since all elements share the same type. Line 28-32 shows an example of checking the addition of two Series objects with the magic method technique in Python, and Line 34-37 shows the apply operation. Note that if the function type information cannot be known statically, PDChecker will need users to provide them with type hints.

```
1  class DataFrame(Type):
2    index_type: Type
3    columns: Dict[str, Type]
4
5    def merge(self, other, on = None, how = None):
6      # check for the parameter how
7      possible_how = ['inner', 'left', 'right', 'outer']
8      if how and how.val not in possible_how:
9        raise CheckerParamError(how.val, possible_how)
10     # get column labels in the parameter on
11     on_labels = [l.val for l in on.val]
12     # ensure the specified labels exist in both data frames
13     for df in [self, other]:
14       missing = [l for l in on_labels if l not in df.columns]
15       if missing:
16         raise CheckerIndexError(missing, df)
17     # check the types of these columns
18     lcols = [ self.columns[l] for l in on_labels]
19     rcols = [other.columns[l] for l in on_labels]
20     if not all(t1.subtype(t2) for t1, t2 in zip(lcols, rcols)):
21       raise CheckerError('type mismatch')
22     # return the merged one
23     return DataFrame(columns=union(self.columns, other.columns))
24
25 class Series(Type):
26   value_type: Type
27
28   def __add__(self, other):
29     if other is Series:
30       return Series(value = self.value + other.value)
31     else:
32       return Series(value = self.value + other)
33
34   def apply(self, func):
35     par_type, ret_type = extract_type(func)
36     assert self.value_type == par_type
37     return Series(ret_type)
```

FIGURE 6. The type-level classes and parts of their check functions.

B. THE LIMITATION OF PDCHECKER

There are two kinds of columns PDChecker cannot handle well: columns that depend on the rows of data frames, i.e., data records, and columns that depend on the results evaluated in Python. For those determined by data records at runtime, PDChecker has no way to check them unless users provide hints. For example,

```
df.transpose()
```

will transpose rows and columns, which converts rows into columns and makes columns unpredictable. Similarly,

```
df.dropna(axis='columns')
```

drop the columns where at least one value is missing and thus the columns cannot be determined in advance. As a consequence of adopting the static approach, PDChecker has no information about such runtime values. However, for the values assigned in code, PDChecker can obtain and trace type information from the beginning. PDChecker can also benefit from the **read_csv** function in pandas to read data from an external source and then get types, as shown in Figure 7.

```
1 def read_csv(fp):
2     import pandas as pd
3     df = pd.read_csv(fp.val)
4     return DataFrame(_index=df.index.dtype,
5                      _columns=df.dtypes.to_dict())
```

FIGURE 7. Using the read_csv function to retrieve data and get types.

The other case that PDChecker cannot handle properly is the code like:

```
df.columns
    = [c.upper() for c in df.columns]
```

Without evaluating the expression in Python, it is not able to understand the column labels. Note that PDChecker only focuses on type checking for data frames and generic type checking for Python has been already discussed in the related work [30]. If there is any operation that involves the type in Python, PDChecker will need users to provide the type information. The integration with a generic Python type checker might be possible and is included in our future work. On the other hand, type inference based on external data sources is already studied by several research activities, and it is not included in the goal of this approach. For example, Type providers in F# language can provide the type information on external data sources at compile-time [32]. The Frames library on Haskell language uses metaprogramming to infer the type of CSV data. There are also several research and implementations devoted to inferring the type of data in JSON format [33]–[35]. Actually, in the implementation of pandas, type inference is used in some functions like the **read_csv** method [36].

C. THE OVERVIEW OF USAGE

PDChecker is designed as a middle layer in the development of data wrangling. As shown in Figure 8, initially, the flow of developing programs with pandas and Python is ①, ②, and ⑥.

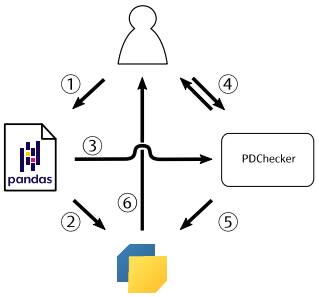


FIGURE 8. The flow of using PDChecker.

After integrating PDChecker into the development, the flow is modified to ①, ③, ④, ⑤, and ⑥. It means PDChecker will check users' code (③) and interact with users (④) before executing the code in Python (⑤). Data scientists can statically check the columns for their data wrangling programs with PDChecker.

For the four problems discussed in the motivating example, Figure 2, we can resolve them as follows:

(*a*) *The labels might not exist in the data frames specified for the operations.*
The labels `'x'`, `'y'`, and `'z'` specified in methods such as **`drop_duplicates`**, **`merge`**, **`groupby`**, and **`agg`** can be correctly checked since PDChecker retrieves the literal values of labels for type checking.

(*b*) *The function parameter string might not be valid.*
Similarly, PDChecker can check the literals used in the parameter of functions, for example **`how`** = `'left'`.

(*c*) *The specified data frames might not have common columns.*
PDChecker prepared type-level classes for data frames, which makes it possible to check whether there are common columns or not.

(*d*) *The function given for the operation might not have the same type as the target column's type.*
With users' type hints, PDChecker can ensure that the type of function given to **`agg`** is consistent with the type of the target column `'z'`.

To improve the usability of such a static checker, we further used the pygls library [37] to benefit from Language Server Protocol[2] (LSP). It is a protocol between editors and language servers providing features like code completion and syntax highlighting, making PDChecker easy to integrate into many environments such as JupyterLab, Visual Studio Code, Eclipse IDE, and even Vim. Figure 9 is a screenshot of using PDChecker to check column labels in JupyterLab.

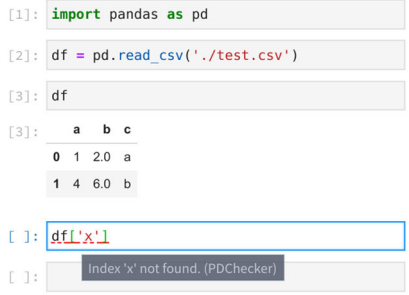


FIGURE 9. Using PDChecker to check column labels in JupyterLab.

V. EVALUATION

This section evaluates our proposal and PDChecker from the functionality of type checking for data frames and compares it with existing implementations. In addition, we provide a user scenario to show how PDChecker can help data scientists in programming data wrangling concretely.

TABLE 1. The functionality of type checking implemented in PDChecker.

operation	check	examples in pandas API	PDChecker
CREATION	FI	pd.read_csv(file)	△ [1]
	FI	pd.DataFrame(list-of-list, columns)	○
	FI	pd.DataFrame(dict-of-list)	○
	FI	pd.Series(list)	○
PROJECTION	FI	df['x']	○
	FI	df.x	○
	FI	df[['x']]	○
	FI	df.loc[:, ['x']]	○
SELECTION	FI	df[df['x'] > 0]	○
	FI, FN	df.query(...)	×
ASSIGNMENT	FI	df['x'] = series-or-list	○
	FI	df.loc[:, ['x']] = series-or-list	○
UNION	FI	pd.concat(df1, df2)	△
JOIN	FI	df.merge(df2, on)	○
	FI, FL	df.merge(df2, on, how)	○
GROUPBY	FI	df.groupby('x', as_index=False)	○
	FI	df.groupby('x', as_index=True)	× [2]
	FI	df.groupby(['x', 'y'], as_index=True)	× [2]
APPLY/AGG	FN	df['x'].apply(lambda-func)	×
	FN	df['x'].apply(func)	△ [1]
	FN	GroupBy.agg(dict-of-func)	○
	FN	GroupBy.agg(list-of-func)	× [2]
MELT/PIVOT	FI	df.pivot('idx', 'col', 'val')	△ [1]
	FI	df.melt(id_vars, value_vars)	○
OPERATIONS	FI	df.x + df.y	○

○ has implementation and supports checking. △ partially implemented and supports checking.
× no implementation or no checking support. [1] Additional type information is necessary.
[2] Unable to check due to the lack of implementing multi-level index.

A. THE FUNCTIONALITY OF TYPE CHECKING FOR DATA FRAMES

Since PDChecker is targeted at checking the errors caused by incorrect references to the columns in data frames, we evaluated the functionality of type checking for the standard operations on data frames. Here we followed the usual operations discussed in the research conducted by Petersohn *et al.* [27] and listed in Table 1. Note that we evaluate proof-of-concept

[2] https://microsoft.github.io/language-server-protocol/

implementation with the standard operations on data frames rather than the rich pandas API.

We classified the abilities of PDChecker to check the potential problems in these operations into three groups: checking the column labels (FI), checking the flag arguments (FL), and checking the function type (FN). FI denotes the ability to check the column labels, i.e., the fields on the given data frame objects. Column labels are necessary input in the operations such as CREATION, PROJECTION, UNION, GROUPBY, and MELT/PIVOT, so a static checker must be able to check whether the specified data frame object has the column (field) or not. FL denotes the ability to check the flag arguments specified in the operations like JOIN. Since the **`how`** parameter set for the methods **`join`** and **`merge`** in pandas should be **`'inner'`**, **`'outer'`**, **`'left'`** or **`'right'`**, a static checker must be able to know whether the given flag arguments are correct or not. Finally, FN denotes the ability to check the type of function specified in the operations SELECTION and APPLY/AGG. Since the specified function will be applied to the data, a static checker needs to figure out whether the function type matches the data type or not.

Our implementation has covered most operations to show the feasibility of our approach. Note that we currently only implement the **`read_csv`** method for external data sources since it is the most used method for loading data. For ASSIGNMENT, we implemented the simplest ones to show its feasibility. Although our implementation currently does not cover multi-level index, advanced functionality in pandas, it is possible to describe with tuples in our approach. However, even implementing multi-level index, stack, and unstack operations cannot be handled due to the limitation of the static check.

B. COMPARING FUNCTIONALITY WITH EXISTING WORK

Here we discuss the implementations of data frames that can trace and check column labels and types so far as we know. The Frames library [7] is built on Haskell, which provides a type-safe API for working with data frames in CSV format. With the help of a powerful type system in Haskell, Frames can represent labels with types, use type-level programming to describe API for data frames, and benefit from metaprogramming to generate types for external data sources. Note that in Frames, some methods such as **`apply`** and **`groupby`** are supported through other libraries rather than Frames itself. The frameless library [8] is a Scala library for improving the expressive ability of types in Spark API [38]. It also benefits from a powerful type system and uses a lot of type-level programming. However, several operations in frameless that change columns will lose column labels since it uses case classes and tuples to represent. Gamma [9] is a language based on objects and nominal typing. It represents external data sources and query operations with objects and methods, respectively. Since Gamma provides types with the methods on objects, it allows users to compose rich and type-safe queries.

TABLE 2. Comparing the ability of type checking with existing work.

operation	pandas	PDChecker*	Frames	frameless	Gamma
CREATION (read CSV)	×	○	○	○	○
CREATION (dynamically)	×	○	○	○	×
SELECTION	×	○	○	○	△
PROJECTION	×	○	○	△	○
UNION	×	△	× *	△	×
JOIN	×	○	○	○	×
GROUPBY	×	○	× *	○	○
APPLY/AGG	×	○	× *	○	×
MELT/PIVOT	×	○	△	×	×

○ has implementation and supports checking. △ partially implemented and supports checking.
× no implementation or no checking support. *Must be used with other libraries.

The differences in the functionality between these implementations and PDChecker are summarized in Table 2. Note that we left pandas in this table for comparison, and this table shows the ability to perform type checking for these operations rather than perform these operations. In other words, it highlights the type checking ability of our approach in the general operations for data frames [27]; the advantages of each implementation over PDChecker are not shown in the table. Unlike other implementations, PDChecker is a standalone checker that must be used along with pandas. Moreover, other implementations use the type system in the language itself, but PDChecker uses abstract interpretation to check the type.

```
df  = pd.read_csv(...)
df1 = pd.read_csv(...)

def some_transform(x: int) -> int: ...

df['b'] = df['b'].apply(some_transform)
df2 = (df
       .merge(df1, on='a')
       [['a', 'b', 'c']])
```

FIGURE 10. An example of reusing code snippets for data wrangling.

C. A USER SCENARIO

This section demonstrates how PDChecker can help data scientists in a typical scenario of data wrangling. For example, suppose that a data scientist wrote a code snippet for data wrangling before, as shown in Figure 10. Now the data scientist got updated data and is going to reuse the code snippet for the data wrangling task at hand. However, there might be some differences in the label and type of columns in the updated data. In an environment like JupyterLab, PDChecker can figure out the errors without executing the code. Thus, there are several advantages: no need to wait for the execution, no side effects in the current execution environment, and meaningful information for debugging.

A) *Warning about the inconsistency between the specified function and column*
In this example, PDChecker can notify the data scientist that the type of **`some_transform`** function does not match the type of column **`'b'`** based on

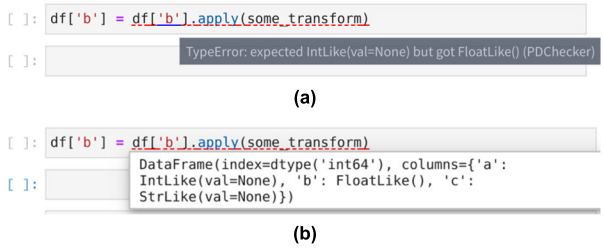


FIGURE 11. The type error (a) and type information (b) given by PDChecker.

the external data source in Line 6 of Figure 10, and shows an error message to warn the data scientist as shown in Figure 11(a). The data scientist can also see the type information of this data frame, as shown in Figure 11(b). Note that PDChecker uses different background colors for tooltips to distinguish errors and information. According to the type information provided by PDChecker, the data scientist can know that the type of column `'b'` became **`float`** this time and make appropriate modifications to the code.

B) Helping in clarifying the root cause of an error after the merge operation

Suppose that the data scientist has fixed the type of some_transform function and continues with the data wrangling task. As shown in Figure 12(a), this time another error marked with a red dotted line appears in the code calling the **`merge`** method, i.e., Line 7-9 of Figure 10. According to the information provided by PDChecker, the data scientist can know that there is no column named `'c'` as expected after the merge operation and the root cause is the duplicate `'c'` columns. Without PDChecker, the data scientist cannot get the error until executing the code as shown in Figure 12(b). Moreover, error messages due to columns are usually not as clear as the information provided by PDChecker.

FIGURE 12. The information given by PDChecker (a) and the runtime error raised in the execution (b).

VI. RELATED WORK

Several known techniques can be used to check data frame columns, but they require language support. Type systems can help programmers to trap type errors. Dependent type [39] was developed to involve values in types. The Π type denotes a function whose return type varies with its arguments, i.e., dependent function type. The Σ type enables an ordered pair where the type of the second element is dependent on the value of the first one, i.e., dependent pair type. In a programming language supporting dependent type, it is possible to represent the relations between the labels and values in the columns of data frames. However, Python does not support dependent types. Row polymorphism [40] lets operations on records be polymorphically typed. In other words, we can define a polymorphic record type for a finite set of field labels, i.e., rows, and it allows programmers to operate on only partial fields of a record. The use of row polymorphism can help the representation of adding/removing columns of data frames. Unfortunately, Python does not support row polymorphism, either. Type-level programming such as introducing type families [41] makes it possible to write functions on types. Type-level functions can represent the operations on data frames well. Although such overloading of data types can be achieved in Haskell, it is still a challenge to do that within languages like the current version of Python.

Domain-specific languages (DSLs) are mini-languages that offer expressive power focused and usually restricted to a particular domain [42]. In general, DSLs can be classified into standalone DSLs and embedded DSLs [43]. Embedded DSLs are those implemented as libraries on a host language and can benefit from the software tools for the host language [44]. Contrarily, the ability of the embedded DSLs greatly depends on the power of their host languages. Data frame libraries such as pandas can be regarded as some sort of embedded DSLs for data frames on top of Python [45], and its ability to check types is limited to the type system of Python. On the other hand, it is possible to create a new standalone DSL for data frames as well by following the design of pandas. However, reimplementing the functionality of a sophisticated and widely-used library might not be a meaningful job. In addition, we need to implement the fundamental parts at the language level rather than benefiting the powerful type system in existing languages such as Haskell and Scala. Moreover, current ecosystems for Python, including tools and libraries, cannot be directly reused.

VII. CONCLUSION

Due to the lack of language support, the columns in data frame libraries cannot be regarded as types, resulting in runtime errors. We proposed statically checking the label and type of columns for data frame libraries. Our idea is based on abstract interpretation but focuses on data frames, which can help programmers find errors due to missing column labels or inconsistent types in data frame operation. We defined the type for data frames and described semantics based on it for type checking. Instead of designing a new language, a standalone checker can be built based on this approach for existing data frame libraries. We took the pandas library as an example to implement a proof-of-concept, PDChecker, to explain implementation details, discuss the limitation, and show the usage concretely. We compared PDChecker with existing work, and the results showed that PDChecker is competitive. Finally, a user scenario was given to demonstrate how PDChecker can be integrated into data wrangling tasks. Based on our research findings, formal verification for the use of data frame libraries can be further studied, and data science software development can benefit from such a checker in practice. Our future directions include extending the semantics to support more data frame operations and integrating with existing generic type checkers.

ACKNOWLEDGMENT

The authors would like to thank the editors and the anonymous reviewers for their valuable comments, which greatly helped improve the quality of this article.

REFERENCES

[1] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, and P. Buono, "Research directions in data wrangling: Visualizations and transformations for usable and credible data," *Inf. Visualizat.*, vol. 10, no. 4, pp. 271–288, Oct. 2011, doi: 10.1177/1473871611415994.

[2] I. G. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino, "Data wrangling: The challenging yourney from the wild to the lake," *Proc. 7th Biennial Conf. Innov. Data Syst. Res. (CIDR)*, Asilomar, CA, USA, Jan. 2015. [Online]. Available: http://cidrdb.org/cidr2015/program.html and http://people.cs.uchicago.edu/~aelmore/class/topics17/wrangling-wild.pdf

[3] B. C. Boehmke, *Data Wrangling With R*. Cham, Switzerland: Springer, 2016. [Online]. Available: https://link.springer.com/content/pdf/bfm%3A978-3-319-45599-0%2F1.pdf

[4] J. M. Chambers and T. J. Hastie, *Statistical Models in S*. Boca Raton, FL, USA: CRC Press, 1991.

[5] R. Ihaka and R. Gentleman, "R: A language for data analysis and graphics," *J. Comput. Graph. Statist.*, vol. 5, no. 3, pp. 299–314, 1996.

[6] W. McKinney, "Data structures for statistical computing in Python," in *Proc. 9th Python Sci. Conf.*, Austin, TX, USA, vol. 445, 2010, pp. 51–56.

[7] A. Cowley. *Frames: Data Frames For Working with Tabular Data Files*. [Online]. Available: https://hackage.haskell.org/package/Frames Accessed: Jan. 23, 2022.

[8] The Frameless Project. *Typelevel/Frameless: Expressive Types for Spark*. Accessed: Jan. 23, 2022. [Online]. Available: https://github.com/typelevel/frameless

[9] T. Petricek, "Data exploration through dot-driven development," in *Proc. 31st Eur. Conf. Object-Oriented Program. (ECOOP)*, 2017, pp. 21:1–21:27. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2017/7261/

[10] Kaggle Inc. *2021 Kaggle Machine Learning & Data Science Survey*. Accessed: Jan. 23, 2022. [Online]. Available: https://www.kaggle.com/c/kaggle-survey-2021/overview

[11] B. Hayes. *For Data Professionals, Python Remains Top Programming Language While R Continues to Decline*. Accessed: Jan. 23, 2022. [Online]. Available: http://businessoverbroadway.com/2021/01/11/for-data-professionals-python-remains-top-programming-language-while-r-continues-to-decline/

[12] Python Software Foundation. *PEP 8—Style Guide for Python Code*. Accessed: Jan. 23, 2022. [Online]. Available: https://www.python.org/dev/peps/pep-0008/

[13] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in Python," *IEEE Access*, vol. 8, pp. 89497–89509, 2020, doi: 10.1109/ACCESS.2020.2990567.

[14] K. J. Millman and M. Aivazis, "Python for scientists and engineers," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 9–12, Mar./Apr. 2011, doi: 10.1109/MCSE.2011.36.

[15] T. E. Oliphant, "Python for scientific computing," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 10–20, May 2007, doi: 10.1109/MCSE.2007.58.

[16] T. E. Oliphant, *A Guide to NumPy*. Trelgol Publishing USA, 2006.

[17] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, 2011.

[18] J. D. Hunter, "Matplotlib: A 2D graphics environment," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 90–95, May 2007, doi: 10.1109/MCSE.2007.55.

[19] F. Perez and B. E. Granger, "IPython: A system for interactive scientific computing," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 21–29, May 2007, doi: 10.1109/MCSE.2007.53.

[20] The pandas Development Team. *Comparison With SQL*. Accessed: Jan. 23, 2022. [Online]. Available: https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_sql.html

[21] Y. Wu, "Is a dataframe just a table?" in *Proc. 10th Workshop Eval. Usability Program. Lang. Tools (PLATEAU)*, 2020, pp. 6:1–6:10. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/11960/

[22] E. J. Ma, Z. Barry, S. Zuckerman, and Z. Sailer, "Pyjanitor: A cleaner API for cleaning data," in *Proc. 18th Python Sci. Conf.*, 2019, pp. 50–53. [Online]. Available: http://conference.scipy.org/proceedings/scipy2019/

[23] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang. (POPL)*, 1977, pp. 238–252.

[24] P. Cousot, "Types as abstract interpretations," in *Proc. 24th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, 1997, pp. 316–331.

[25] The pandas Development Team. *Pandas.DataFrame—Pandas 1.1.0 Documentation*. Accessed: Jan. 23, 2022. [Online]. Available: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

[26] Python Software Foundation. *PEP 484—Type Hints*. Accessed: Jan. 23, 2022. [Online]. Available: https://www.python.org/dev/peps/pep-0484/

[27] D. Petersohn, S. Macke, D. Xin, W. Ma, D. Lee, X. Mo, J. E. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. Parameswaran, "Towards scalable dataframe systems," 2020, *arXiv:2001.00888*.

[28] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, doi: 10.1145/1327452.1327492.

[29] P. Jupyter. *JupyterLab and Jupyter Notebook*. Accessed: Jan. 23, 2022. [Online]. Available: https://jupyter.org

[30] R. Monat, A. Ouadjaout, and A. Miné, "Static type analysis by abstract interpretation of Python programs," in *Proc. 34th Eur. Conf. Object-Oriented Program. (ECOOP)*, 2020, pp. 17:1–17:29. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13174/

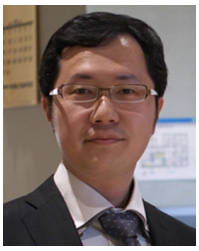
[31] A. Fromherz, A. Ouadjaout, and A. Miné, "Static value analysis of Python programs by abstract interpretation," in *Proc. NASA Formal Methods Symp.* Newport News, VA, USA: Springer, 2018, pp. 185–202. [Online]. Available: https://hal.sorbonne-universite.fr/hal-01782390/document

[32] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek, "Themes in information-rich functional programming for internet-scale data sources," in *Proc. Workshop Data driven Funct. Program. (DDFP)*, Rome, Italy, 2013, pp. 1–4, doi: 10.1145/2429376.2429378.

[33] T. Petricek, G. Guerra, and D. Syme, "Types from data: Making structured data first-class citizens in F#," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Santa Barbara, CA, USA, Jun. 2016, pp. 1–14, doi: 10.1145/2908080.2908115.

[34] The Quicktype Community. *Quicktype/Quicktype*. Accessed: Jan. 23, 2022. [Online]. Available: https://github.com/quicktype/quicktype

[35] M. J. Gajda. *JSON-Autotype: Automatic Type Declaration for JSON Input Data*. Accessed: Jan. 23, 2022. [Online]. Available: https://hackage.haskell.org/package/json-autotype
[36] A. Golubin. *How Pandas Infers Data Types When Parsing CSV Files*. Accessed: Jan. 23, 2022. [Online]. Available: https://rushter.com/blog/pandas-data-type-inference/
[37] Open Law Library. *Openlawlibrary/Pygls*. Accessed: Jan. 23, 2022. [Online]. Available: https://github.com/openlawlibrary/pygls
[38] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Melbourne, VIC, Australia, May 2015, doi: 10.1145/2723372.2742797.
[39] P. Martin-Löf, "An intuitionistic theory of types: Predicative part," in *Studies in Logic and the Foundations of Mathematics*, vol. 80, H. E. Rose and J. C. Shepherdson, Eds. Amsterdam, The Netherlands: Elsevier, 1975, pp. 73–118.
[40] M. Wand, "Type inference for record concatenation and multiple inheritance," *Inf. Comput.*, vol. 93, no. 1, pp. 1–15, 1991, doi: 10.1016/0890-5401(91)90050-C.
[41] O. Kiselyov, S. P. Jones, and C.-C. Shan, "Fun with type functions," in *Reflections on the Work*, C. A. R. Hoare, A. W. Roscoe, C. B. Jones, K. R. Wood, Eds. London, U.K.: Springer, 2010, pp. 301–331.
[42] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN Notices*, vol. 35, no. 6, pp. 26–36, Jun. 2000, doi: 10.1145/352029.352035.
[43] P. Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4es, p. 196, Dec. 1996.
[44] P. Hudak, "Modular domain specific languages and tools," in *Proc. 5th Int. Conf. Softw. Reuse*, Jun. 1998, pp. 134–142, doi: 10.1109/ICSR.1998.685738.
[45] A. Andrzejak, K. Kiefer, D. E. Costa, and O. Wenz, "Agile construction of data science DSLs (tool demo)," in *Proc. 18th ACM SIGPLAN Int. Conf. Generative Program., Concepts Experiences*, Athens, Greece, Oct. 2019, pp. 27–33, doi: 10.1145/3357765.3359516.



YUNGYU ZHUANG (Member, IEEE) received the B.S. and M.S. degrees in mechanical engineering and computer science from the National Taiwan University, Taiwan, in 2002 and 2004, respectively, and the Ph.D. degree in information science and technology from The University of Tokyo, Japan, in 2014. He was a Research Assistant with the Central Weather Bureau, Taiwan, from 2004 to 2006. From 2006 to 2011, he worked as a software engineer at the industry. He is currently an Assistant Professor with the Department of Computer Science and Information Engineering, National Central University. From 2014 to 2016, he was a Project Assistant Professor with The University of Tokyo. His research interests include programming language design, software engineering, high-performance computing, machine learning, and programming education

MING-YANG LU received the B.S. degree in computer science and information engineering from Feng Chia University, in 2016, and the M.S. degree in computer science and information engineering from the National Central University, Taiwan, in 2020. His research interests include programming languages, software engineering, and data science.