Static Analysis of Data Transformations in Jupyter Notebooks

Luca Negrini
luca.negrini@corvallis.it
Corvallis Srl
Verona, Italy

Guruprerana Shabadi
guruprerana.shabadi@polytechnique.edu
École Polytechnique
Institut Polytechnique de Paris
Paris, France

Caterina Urban
caterina.urban@inria.fr
Inria & ENS | PSL, France
Paris, France

Abstract

Jupyter notebooks used to pre-process and polish raw data for data science and machine learning processes are challenging to analyze. Their data-centric code manipulates dataframes through call to library functions with complex semantics, and the properties to track over it vary widely depending on the verification task. This paper presents a novel abstract domain that simplifies writing analyses for such programs, by extracting a unique CFG from the notebook that contains all transformations applied to the data. Several properties can then be determined by analyzing such CFG, that is simpler than the original Python code. We present a first use case that exploits our analysis to infer the required shape of the dataframes manipulated by the notebook.

CCS Concepts: • **Theory of computation → Program analysis**; **Abstraction**; • **Software and its engineering → Automated static analysis**.

Keywords: Static Analysis, Abstract Interpretation, Data Science, Jupyter Notebooks

ACM Reference Format:
Luca Negrini, Guruprerana Shabadi, and Caterina Urban. 2023. Static Analysis of Data Transformations in Jupyter Notebooks. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3589250.3596145

1 Introduction

The ever-increasing usage of data-driven decision processes led to *data science* (DS) and *machine learning* (ML) permeating several areas of everyday life, reaching outside the boundaries of computer science and software engineering. Ensuring correctness of these processes is particularly important when they are employed in critical areas like medicine, public policy, or finance. In contrast to robustness verification of trained ML models [9], data pre-processing of the DS/ML has received little attention. As raw data is often inconsistent or incomplete, pre-processing programs, typically implemented in Jupyter notebooks, apply transformations to polish it to the point where it can be visualized or used for training. Errors and inconsistencies at this stage can silently propagate downwards in the DS/ML chain, leading to incorrect conclusions and below-par models [1].

Verification of notebooks can take different directions, such as detecting data leakages (i.e., sharing of information between the training and test datasets [7]) or warning about the introduction of bias or skews [11]. Regardless, Jupyter notebooks are challenging to analyze: code comes in blocks that can be executed in any order with repetitions, and data is manipulated through calls to a vast number of library functions with complex and possibly overlapping semantics.

This paper proposes an abstract interpretation approach to simplify the implementation of verification techniques for Jupyter notebooks containing DS/ML programs. We propose an abstract domain that tracks transformations made to *dataframes*, that is, the in-memory tables containing the input data, in a unique graph. The latter contains data transformations as nodes that are linked by edges encoding the order in which they are applied. The final graph produced at the end of the analysis is a control flow graph (CFG) containing only dataframe transformations: analyses such as the ones mentioned above can be implemented as fixpoints over this CFG, instead of tackling the more complex Python code.

This paper is structured as follows. Section 2 discusses related work. Section 3 introduces PyLiSA, the analyzer used to evaluate our domain, and LiSA, the framework it relies on. Section 4 defines the abstraction for dataframe values. We then explore a first use-case in Section 5, where we use our abstraction for inferring the shape of the dataframes used by a program. We then conclude with a preliminary experiment on a real DS notebook in Section 6.

2 Related Work

Obtaining formal guarantees on the safety and fairness of ML models has been a subject of recent widespread interest [9].

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06…$15.00
https://doi.org/10.1145/3589250.3596145

Our work builds upon this large ecosystem and proposes a verification framework for the data pre-processing stage of the ML pipeline. The closest body of work is [7] which also analyses DS notebooks along with their peculiar execution semantics and proposes an abstraction to detect data leakage. Our approach towards the shape inference of input data follows the work of [8] and extends it to support inputs to programs which contain datasets. Similarly, our objective of inferring input data usage directly derives from the compound data structure usage analysis presented in [10] and adds the ability to track the usage of selections of datasets. The objective of detecting bias/skew introduction is inspired from the `mlinspect` tool proposed in [4]. This tool builds a directed acyclic graph (DAG) of operations (like filters or projections) applied to the data by analyzing the code and using framework-specific backends (like `scikit-learn`). After analyzing the DAG, it suggests potential sources of bias/skew. Although this is promising, it only places syntactic checks and cannot concretely detect which operations cause these problems. Lastly, [5] is an automated data provenance system for Python. However, it requires executing the code which is not always be feasible when large datasets are involved.

3 LiSA and PyLiSA

LiSA [3, 6] (**Li**brary for **S**tatic **A**nalysis) is a modular framework for developing static analyzers based on the abstract interpretation theory. LiSA analyzes CFGs whose statements do not have predefined semantics: instead, users of the framework define custom statement instances implementing language-specific semantic functions, enabling the analysis of a wide range of programming languages and the development of multilanguage analyses. The analysis infrastructure is partitioned into three main areas: call evaluation, memory modeling and value analysis. Each area corresponds to a separate analysis component, that operates agnostically w.r.t. how the others are implemented. At first, calls are abstracted by the *Interprocedural Analysis*, that leaves the remaining components with call-less programs. Then, memory-related expressions are abstracted by the *Heap Domain*, yielding call- and memory-less programs for the *Value Domain* to analyze. Code parsing and semantics are defined in *Frontends*, that can also provide implementations for LiSA's components. In this paper, we employ the Python frontend PyLiSA[1].

4 The Dataframe Graph Domain

We present an abstraction able to capture the structure of the dataframes manipulated in Python code. Intuitively, we employ a graph structure to keep track of all operations that involve dataframes, with edges encoding the order in which they are performed. The graph thus represents the state of each dataframe at a given program point. Nodes of this graph can be referenced by variables of the program. The latter thus refers to the dataframe corresponding to the sub-graph obtained with a backward DFS starting from the node. We adopt a two-level mapping: program variables point to labels, and the latter are mapped to the nodes. This enables simple handling of dataframe aliasing (i.e., two variables will be mapped to the same label) and updates (i.e., by changing the nodes pointed by a label, we indirectly update all variables pointing to the label).

Our domain $\mathcal{D}^\#$ is meant to be an abstraction of the portion $\mathcal{D}$ of the program state that stores information about dataframes: we rely on an auxiliary domain to reason about the remainder of the state. As, in our experience, non-dataframe values appearing in the notebooks we target are mostly constants, $\mathcal{D}^\#$ cooperates with a simple constant propagation domain $\mathcal{CP}^\#$. We rely on the semantics of $\mathcal{CP}^\#$ to abstract:

- a string expression s as a constant string σ;
- a list of strings cl as a constant list of constant strings $\langle \sigma_1, \ldots, \sigma_n \rangle$;
- a list of dataframes dl as a constant list of abstract labels $\langle \ell_1, \ldots, \ell_n \rangle$ that will be introduced shortly;
- the left-hand side of an assignment x to a set of identifiers $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$.

We begin defining $\mathcal{D}^\#$ by introducing the graph structure, where a graph $g^\# = (N, E) \in \mathcal{G}^\#$ is composed by a set of nodes $N \subseteq \mathcal{N}$ and a set of edges $E \subseteq \mathcal{E}$. Elements of $\mathcal{N}$ are:

- `read`(σ), initializing a dataframe with the contents of file σ;
- `access`$(\sigma_1, \ldots, \sigma_k)$, accessing columns $\sigma_1, \ldots, \sigma_k, k \in \mathbb{N}$;
- `transform`(f), transforming values through an auxiliary function f;
- `concat`, concatenating multiple dataframes;
- `filter`(σ, op, v), selecting rows where v^σ op v holds (with v^σ being the value of column σ);
- `assign`$(\sigma_1, \ldots, \sigma_k)$, assigning columns $\sigma_1, \ldots, \sigma_k, k \in \mathbb{N}$ to a new value.

where σ and v are string and value abstractions in $\mathcal{CP}^\#$, $op \in \{=, \neq, >, \geq, <, \leq\}$ and f is the signature of a Python function. Instead, elements of $\mathcal{E}$ are (with $n, n' \in \mathcal{N}, i \in \mathbb{N}$):

- $n \rightarrow n'$ is an edge encoding the sequential order of operations;
- $n \rightsquigarrow_i n'$ is a concatenation edge, where n is the i-th dataframe in the concatenation that builds n' (note that n' can have more incoming concatenation edges using the same i, indicating multiple candidates for the same index);
- $n \twoheadrightarrow n'$ is an assign edge, where n is the right-hand side of the assignment n' (once more, n' can have more incoming assign edges, indicating multiple candidates for the right-hand side).

$\mathcal{D}^\#$ also contains two maps, both relying on abstract labels. A label $\ell \in \mathcal{L}$ is an arbitrary synthetic identifier that serves

[1] https://github.com/lisa-analyzer/pylisa.

```
1 import pandas as pd
2 df1 = pd.read_csv('italy.csv')
3 df1['birth'] = pd.to_datetime(df1['birth'])
4 df2 = pd.read_csv('france.csv')
5 df2 = df2['age' < 50]
6 df3 = pd.concat([df1, df2])
```

Figure 1. Python DS running example

as an abstract name for a set of nodes in $\mathcal{N}$, where $\mathscr{L}$ is the finite set of all possible labels. While we do not impose any specific structure on $\mathscr{L}$, a common characterization of labels is to have one for each program point. $\mathcal{D}^\#$ contains (i) a function $\mathscr{L} \to \wp(\mathcal{N}) \in \mathcal{L}^\#$ from labels to sets of nodes, and (ii) a map $\mathbb{V}ar \to \wp(\mathscr{L}) \in \mathcal{V}^\#$ keeping track of which possible labels a variable can refer to. Notice that, depending on the analyzer's infrastructure, variables can correspond to abstract memory locations or program variables.

We can now define $\mathcal{D}^\#$ as the Cartesian product $\mathcal{V}^\# \times \mathcal{L}^\# \times \mathcal{G}^\#$, that is a complete lattice since $\mathcal{G}^\#$ is a Cartesian product of powersets, and $\mathcal{V}^\#$ and $\mathcal{L}^\#$ are functional lifts of powersets. One concern with infinite lattices such as $\mathcal{D}^\#$ is the convergence of fixpoint iterations over them. As $\mathcal{G}^\#$ intuitively does not satisfy ACC[2], a widening operator is required. As, in our experience, the DS notebooks that this domain targets mostly contain sequential code with very few loops that stabilize in few iterations, we employ a naive widening as $d_1^\# \nabla d_2^\# = d_1^\#$ **if** $d_1^\# = d_2^\#, \top$ **otherwise**. With such an operator we ensure termination of the analysis, and we leave the study of a more precise widening as future work.

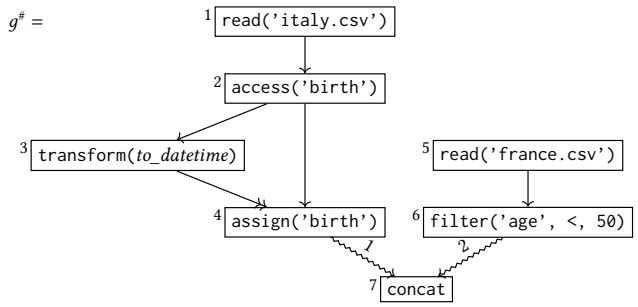
Example. Figure 2 reports the $d^\#$ instance abstracting the code of Figure 1. For the sake of clarity, nodes of $g^\#$ are enriched with a numerical identifier on the top-left corner to easily identify them. Such identifiers are used in the co-domain of $l^\#$ to represent them. We show how this graph is constructed while defining the abstract semantics.

The connection between $\mathcal{D}$ and $\mathcal{D}^\#$ is established by the abstraction function α and the concretization function γ. A set of functions $\{\bar{d}_1, \ldots, \bar{d}_m\}, \bar{d}_i \in \mathcal{D}, 1 \leq i \leq m, m \in \mathbb{N}^\infty$ can be abstracted to an element $d^\# \in \mathcal{D}^\#$ through function $\alpha : \wp(\mathcal{D}) \to \mathcal{D}^\#$, defined as the lub of the abstractions of each individual $\bar{d}$. The abstraction of a single function is defined as $\alpha(\bar{d}) = (v^\#, l^\#, g^\#)$, with:

- $\bar{d} = \{(v_1, d_1), \ldots, (v_k, d_k)\}, 1 \leq i \leq k, k \in \mathbb{N}$;
- $(g_i^\#, n_i) = \mathsf{shape}(d_i)$, $g^\# = \bigcup_i g_i^\#$;
- $l^\# = \{(\ell_1, \{n_1\}), \ldots, (\ell_k, \{n_k\})\}$;
- $v^\# = \{(v_1, \{\ell_1\}), \ldots, (v_k, \{\ell_k\})\}$.

The abstraction of a single dataframe map exploits $\mathsf{shape} : \mathbb{D} \to \mathcal{G}^\# \times \mathcal{N}$, an auxiliary function that extracts the shape of a concrete dataframe as a single-path graph containing only a `read` node followed by an `access` node reporting all the existing columns, and returning the graph itself and its

[2]As $\mathcal{N}$ and $\mathscr{E}$ are infinite sets, one can keep adding new nodes and edges without the graph ever stabilizing.



$$v^\# = \{(\mathtt{df1}, \{\ell_1\}), (\mathtt{df2}, \{\ell_4\}), (\mathtt{df3}, \{\ell_5\})\}$$
$$l^\# = \{(\ell_1, \{4\}), (\ell_2, \{3\}), (\ell_3, \{5\}), (\ell_4, \{6\}), (\ell_5, \{7\})\}$$

Figure 2. Example $d^\#$ abstracting the code of Figure 1

unique leaf. The abstraction of a set of states is thus the union of the abstraction of each individual state, generated by creating the graph $g^\#$ containing the shape (that is, the access to all columns C optionally preceded by the reading of source S) of all existing dataframes, having each variable refer to the corresponding node in $g^\#$.

As α is *join-preserving*, the concretization function γ can be defined in terms of α, according to Proposition 7 of [2], also inducing the Galois connection $\langle \wp(\mathcal{D}), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}^\#, \sqsubseteq \rangle$, where $\sqsubseteq$ is the lift of $\subseteq$ to functions and Cartesian products.

Abstract Semantics. The abstract semantics of $\mathcal{D}^\#$ is defined w.r.t the one of $\mathcal{CP}^\#$, that is used to evaluate non-dataframe expressions. $\mathcal{D}^\#$ evaluates dataframe expressions to a set of labels, identifying nodes representing the dataframes that correspond to the expression. In the following, we give intuitive definitions of the semantics of expressions that involve dataframes.

Assignment. Whenever the right-hand side of an assignment `x = df` is a dataframe expression, $v^\#$ must be updated for the corresponding variable. Specifically, as `df` evaluates to a set of labels $\{\ell_1, \ldots, \ell_w\}$, and `x` evaluates to a set of identifiers $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$, $v^\#$ can be updated to $v' = v'[\mathtt{x}_i \mapsto \{\ell_1, \ldots, \ell_w\}], \forall \mathtt{x}_i \in \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$.

Example. When evaluating the assignment at line 2 of Figure 1, the semantics stores the label pointing to the `read` node (whose creation is dictated by the abstract semantics of *read*). $v^\#$ is thus extended with the pair $(\mathtt{df1}, \{\ell_1\})$, where $\{\ell_1\}$ is the label returned by the semantics of *read*.

Variable evaluation. Whenever a variable is referenced throughout the program, our semantics must evaluate it to the corresponding labels if it refers to a dataframe, while the remaining variables are handled by $\mathcal{CP}^\#$. Thus, when `x` resolves to a dataframe, it evaluates to $v^\#(\mathtt{x})$.

Example. When evaluating line 6 of Figure 1, `df1` and `df2` must be first resolved to the dataframes they represent: `df1` evaluates to $\{\ell_1\}$ while `df2` is evaluated to $\{\ell_4\}$, as the two variables are mapped to $\{\ell_1\}$ and $\{\ell_4\}$ in $v^\#$, respectively.

Dataframe initialization. When dataframes are initialized with the contents of an external resource through *read*, the operation cannot be precisely modeled statically as the contents of the resource are unknown at compile time. We thus symbolically record the source of the data by adding a `read` node to the graph. If the argument of *read* is abstracted by $\mathcal{CP}^\#$ as σ, the semantics adds a new `read`(σ) and returns a unique label $\{\bar{\ell}\}$ that points to the freshly added node.

Example. When the `read` call at line 2 of Figure 1 is evaluated, $g^\#$ is still empty. A single `read('italy.csv')` node is added to $g^\#$ by the semantics, that then extends $l^\#$ with a fresh label ℓ_1, that is mapped to a set containing the node itself, and is used as the only label in the result of the evaluation.

Column access. The *access* transformation accesses a set of columns of the target dataframe. As this operation does not create a new dataframe, it is modeled as an in-place operation, directly affecting the nodes pointed by the labels of its argument: the semantics adds an `access`$(\sigma_1, \dots, \sigma_n)$ node (where $\sigma_1, \dots, \sigma_n$ is the $\mathcal{CP}^\#$ abstraction for the column list), connecting it to the nodes of the first argument using normal edges $\rightarrow$ and mapping it to a new label.

Example. When the access to column `birth` at line 3 of Figure 1 is evaluated, `df1` is first processed, producing $\{\ell_1\}$ as abstract value. Then, the evaluation of the column names yields $\langle$`'birth'`$\rangle$, resulting in the creation of node `access('birth')`. The graph is then extended adding (i) the newly created node with id 2, and (ii) a normal edge connecting nodes 1 and 2. Furthermore, ℓ_1 is remapped to $\{2\}$ inside the resulting $l^\#$ (the mapping is not visible in Figure 2 as evaluation of following statements overwrites it).

Value transformation. Tracking transformations of dataframe values can be problematic, as the functions carrying the transformation must be summarized somehow. Instead, we provide a lightweight semantics that records the signature of the used function inside a node of the graph, deferring further reasoning to successive abstractions. As this is not an in-place operation, the semantics creates a new label that is mapped to the transformation node, and the latter is connected to nodes representing the target dataframe (thus branching off the original dataframe). The label is then returned as the unique result of the evaluation. The new `transform`(f) node is also connected to the nodes of the receiving dataframe using normal edges $\rightarrow$.

Example. When the `to_datetime` call at line 3 of Figure 1 is evaluated, our semantics first evaluates the column access, producing $\{\ell_1\}$ as shown earlier. The semantics then (i) creates a unique `transform(to_datetime)` node, (ii) adds it to the graph with id 3, and (iii) connects it to node 2 with a normal edge. The mapping between ℓ_2, a fresh and unused label, and the singleton set containing node 3 is also introduced in $l^\#$, and $\{\ell_2\}$ is returned as the evaluation's result.

Dataframe assignment. When a portion of a dataframe is overwritten with new values, the semantics of variable assignment cannot be employed, as no variable changes value. Instead, the semantics of *assign* records the assignment as an `assign`$(\sigma_1, \dots, \sigma_n)$ node in the graph (with $\sigma_1, \dots, \sigma_n$ is the abstraction of the column list produced by $\mathcal{CP}^\#$), connected to both the dataframe receiving it (using normal edges $\rightarrow$) and the value being stored (using assign edges $\twoheadrightarrow$).

Example. When the assignment at line 3 of Figure 1 is evaluated, `df1` is evaluated to $\{\ell_1\}$, the column names evaluate to the constant list $\langle$`'birth'`$\rangle$, and the *transform* expression is resolved to $\{\ell_2\}$. The semantics proceeds by (i) creating an `assign('birth')` node, (ii) adding it to the graph with id 4, (iii) connecting it to node 3 (image of ℓ_2 in $l^\#$) with an assign edge, and finally (iv) connecting it to node 2 (image of ℓ_1 in $l^\#$) with a normal edge. To conclude, ℓ_1 is remapped to a set containing node 4 in $l^\#$, as the assignment has side-effects.

Rows filtering. Similarly to value transformations, abstracting rows filters can be problematic, as different facts about the filtering conditions can be of interest depending on the target analysis. We thus record the condition as-is within a `filter`(σ, op, v) node, delegating its interpretation to successive abstractions. Note that this is not an in-place operation: original dataframes are not modified, but a filtered view of them is returned by the operation. We reflect this in our formalization by yielding a fresh label pointing to the filtering node. The semantics connects the filter node to the ones of the receiving dataframe with normal edges $\rightarrow$.

Example. When rows are filtered at line 5 of Figure 1 is evaluated, the semantics first evaluates its arguments: `df2` is evaluated to $\{\ell_3\}$, the column name evaluates to `'age'`, and the value used for the comparison to the constant `50`. A `filter('age', <, 50)` node is then created, and it is added the graph with id 6. A normal edge connecting it to node 5 (image of ℓ_3 in $l^\#$) is also introduced, and the label ℓ_4 is created and mapped to a set containing node 6 in $l^\#$.

Concatenation. The semantics of the concatenation must create a new dataframe with the contents of all of its arguments, that come compacted into a list. For of our domain, this means connecting all nodes of each argument to a `concat` node, that will be the image of a new label, using concatenation edges $\rightsquigarrow$ with the corresponding indexes.

Example. When the `concat` call at line 6 of Figure 1 is evaluated, the semantics first evaluates the list of target dataframes, producing $\langle\{\ell_1\}, \{\ell_4\}\rangle$ as abstraction for the elements of the list. The semantics then (i) creates the `concat` node, (ii) adds it to the graph with id 7, and (iii) connects it to nodes 4 and 6 (images of ℓ_1 and ℓ_4 in $l^\#$) with concatenation edges indexed with 1 and 2. A fresh label ℓ_5 is generated, and is mapped to a singleton set containing node 7 in $l^\#$.

5 A First Application: Shape Inference

Tracking dataframe transformations through $\mathcal{D}^\#$ is just the beginning. In general, successive analyses targeting $\mathcal{D}^\#$ instead of starting from the Python code can still be formalized

and implemented as abstract interpretations. Fixpoint algorithms can be applied over instances of $\mathcal{G}^\#$, treating the nodes as code. In this context, one has to define the abstract semantics w.r.t. each node's meaning, possibly taking into account the edges attached to them. In this section, we explore one of the possible objectives in the analysis of DS programs, as we aim at inferring the shape of each dataframe read from an external source.

We start by defining the domain $C^\#$ of columns. Denoting as $\overline{\Sigma} = \Sigma^* \cup \{\top, \bot\}$ the set possible string abstractions provided by $\mathcal{CP}^\#$, elements of $C^\#$ are functions $\wp(\overline{\Sigma}) \to (\wp(\overline{\Sigma}) \times \wp(\overline{\Sigma}))$ whose domain is composed by sets of strings $\{\sigma_1, \ldots, \sigma_p\}$ representing sources of data, and whose co-domain is built over pairs of sets $\{\dot{\sigma}_1, \ldots, \dot{\sigma}_n\}, \{\hat{\sigma}_1, \ldots, \hat{\sigma}_m\}$. Each $\dot{\sigma}_i$ is a column that is accessed before being assigned (and thus must be part of the original dataframe to prevent runtime errors), and each $\hat{\sigma}_i$ is a column that is assigned during the execution (and thus might not exist in the original dataframe). The columns domain thus aims at inferring, for each external source of data, what columns must exist for the program to not crash. In our abstraction, we consider sets of sources as function keys since dataframes can be created through concatenation, and can thus have multiple sources. In this case, all accessed columns must exist in at least one source. Note that $C^\#$ is a complete lattice, as it is a functional lift of a Cartesian product of powersets.

We informally define the semantics of $C^\#$ w.r.t. nodes of $\mathcal{G}^\#$, as this kind of analysis does not need variable information stored in $\mathcal{L}^\#$ and $\mathcal{V}^\#$. The semantics relies on the auxiliary function `sources` : $\mathcal{N} \to \wp(\overline{\Sigma})$ that extracts all sources (i.e., arguments of `read` nodes) whose data is used to build the dataframe identified by the given node.

Read. When reading data from an external resource, no particular column is accessed. Instead, we define an entry in our state corresponding to the possible abstractions of the resource identifier.

Concat. Similarly to `read`, `concat` does not access any column, but instead introduces a new entry in the post-state corresponding to the union of its sources.

Transform. A transformation does not access any column explicitly, as it operates on the entirety of the dataframe that receives the transformation. As such, its semantics is defined as the identity function.

Access. The column access is the main vector for referencing columns by-name. This is reflected by its semantics, that adds every column name to the left-most set of the corresponding sources if we have no evidence of it being defined earlier.

Filter. As rows filtering is expressed as a condition over the value of a specific column, it indirectly represents a column access. This is once more reflected as the addition of the column name to the left-most set of the corresponding sources, if it was not defined before.

Assign. The dataframe assignment is the only node kind that can safely define non-existing columns. The semantics of this node adds the abstract column names to the right-most set of the corresponding sources.

Example. We now use $C^\#$ to infer the shape of the dataframes abstracted by the graph in Figure 2. For the sake of clarity, we first analyze the left-most branch of the graph as a whole, followed by the right-most one, concluding the analysis with the `concat` node. The analysis begins applying the semantics of `read` to node 1, using an empty $C^\#$ instance $c_0^\# = \{\}$ as pre-state, and producing the entry for the read resource $c_1^\# = \{(\{\texttt{'italy.csv'}\}, (\emptyset, \emptyset))\}$. This is in turn used as pre-state for the evaluation of node 2, where the semantics of `access` populates the function with the accessed column, producing $c_2^\# = \{(\{\texttt{'italy.csv'}\}, (\{\texttt{'birth'}\}, \emptyset))\}$. As the semantics of `transform` is the identity function, the pre-state of node 4 is the join of the post-state of both predecessors: $c_2^\# \dot{\cup} c_2^\# = c_2^\#$. When such result is used as argument for the `assign` semantics, it produces the post-state $c_3^\# = \{(\{\texttt{'italy.csv'}\}, (\{\texttt{'birth'}\}, \{\texttt{'birth'}\}))\}$. Equivalently, the analysis starts with the same empty pre-state $c_0^\#$ at node 5, applying the `read` semantics and producing the post-state $c_4^\# = \{(\{\texttt{'france.csv'}\}, (\emptyset, \emptyset))\}$. Then, $c_4^\#$ is used to compute the result of the `filter` semantics at node 6, that is $c_5^\# = \{(\{\texttt{'france.csv'}\}, (\{\texttt{'age'}\}, \emptyset))\}$. Lastly, at node 7, the pre-state is built as the lub of the predecessors' post-states $c_6^\# = c_3^\# \dot{\cup} c_5^\#$ that thus contains both entries. The semantics of `concat` can then be applied to $c_6^\#$, producing the final state of the analysis:

$$c_7^\# = \left\{ \begin{array}{l} (\{\texttt{'italy.csv'}\}, (\{\texttt{'birth'}\}, \{\texttt{'birth'}\})), \\ (\{\texttt{'france.csv'}\}, (\{\texttt{'age'}\}, \emptyset)), \\ (\{\texttt{'italy.csv'}, \texttt{'france.csv'}\}, (\emptyset, \emptyset)) \end{array} \right\}$$

6 An Early Experiment Using PyLiSA

Both $\mathcal{D}^\#$ and $C^\#$ have been implemented in PyLiSA, that analyzes Jupyter notebooks by extracting the Python code from the cells that contains it. Cells are analyzed according to a specific user-defined execution sequence, defaulting to the order in which cells are defined in the notebook.

We provide the semantics of `pandas` library's functions through LiSA's native CFGs. These are special CFGs that contain a single statement. Calls to native CFGs are evaluated by computing the semantics of their unique node, without running additional fixpoints: the node's semantics thus becomes an abstract summary of the modeled function. Specifically, the semantics of native CFGs modeling `pandas` functions converts them to the nodes presented in Section 4. $\mathcal{D}^\#$ has been implemented as a *Value Domain*, directly embedding the constant propagation domain. At the end of the analysis, a $\mathcal{G}^\#$ instance is extracted from the post-state of the last instruction, and a fixpoint is executed over the it using $C^\#$. Warnings are then issued to inform the user about columns accessed and assigned for each data source.

As a first experiment, we selected the *"Coronavirus (COVID-19) Visualization & Prediction"*[3] dataframe, one of the most popular notebooks aggregating data from different sources on Kaggle, a public repository of Jupyter notebooks for DS. The graph produced when analyzing such code is published on a GitHub Gist[4] as it is too large for this manuscript. Note that the implemented analysis supports additional pandas constructs w.r.t. the ones presented in this paper, that have been omitted as they do not contribute further to the intuition behind the domain. In the graph, these take the form of additional node kinds, whose intuitive meaning is explained in the Gist's introduction. The analysis generates the following warnings (where URL of csv files have been trimmed for compactness), correctly identifying all column names that appear in the notebook:
[File: *daily_reports/08-23-2022.csv*] Columns accessed before being assigned: 'Confirmed', 'Province_State', 'Country_Region', 'Incident_Rate', 'Deaths'
[File: *daily_reports_us/08-23-2022.csv*] Columns accessed before being assigned: 'Province_State', 'Testing_Rate', 'Total_Test_Results'
[File: *time_series_covid19_confirmed_global.csv*] Columns accessed before being assigned: 'Country/Region'
[File: *time_series_covid19_deaths_global.csv*] Columns accessed before being assigned: 'Country/Region'

7 Conclusion

This paper presents an abstract interpretation approach to analyze Python programs employed in data science and machine learning. Such programs manipulate dataframes, that is, complex in-memory tables collecting data that can be used to guide decision processes or train machine learning models. We designed an abstract domain that extracts the operations performed over dataframes, building a graph that encodes the order in which they are performed. Such a graph can be the subject of further analyses, inferring several properties such as the *shape* of the dataframes read by the program, or the absence of data leakages between training and testing phases of a machine learning process. As a guiding example of how to exploit our domain, we defined a simple abstract interpretation that computes, for each file read by the source program (and thus present inside the graph), the set of columns that are either accessed before being assigned, or defined through an assignment. We provided an early implementation of both domains in PyLiSA, a LiSA frontend for Python programs.

There are plenty of future directions that our work can take. As this work is still ongoing, the obvious first line of axis is to prove the soundness of the proposed semantics abstractions, followed by an investigation on their completeness to further improve the domain. Besides, inferring the shape of dataframes is not the only useful analysis one can employ for DS software. In fact, after strengthening shape inference to incorporate rows and cell properties, we aim at providing abstractions to detect data leakages and biases. Lastly, we aim at extending the abstraction to more pandas functions, and to further libraries other than pandas itself.

[3] https://www.kaggle.com/code/therealcyberlord/coronavirus-covid-19-visualization-prediction, version 722.
[4] https://gist.github.com/lucaneg/9621f3296b7b47b12c5ee1c52066b3d1.

References

[1] Irene Y. Chen, Fredrik D. Johansson, and David Sontag. 2018. Why is My Classifier Discriminatory?. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3543–3554. https://proceedings.neurips.cc/paper/2018/file/1f1baa5b8edac74eb4eaa329f14a0361-Paper.pdf

[2] Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation and application to logic programs. *The Journal of Logic Programming* 13, 2 (1992), 103–179. https://doi.org/10.1016/0743-1066(92)90030-7

[3] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. 2021. Static Analysis for Dummies: Experiencing LiSA. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Virtual, Canada) *(SOAP 2021)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3460946.3464316

[4] Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian Schelter. 2021. MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2736–2739. https://doi.org/10.1145/3448016.3452759

[5] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 1542–1551. https://doi.org/10.1145/3394486.3403205

[6] Luca Negrini. 2023. *A generic framework for multilanguage analysis*. Ph. D. Dissertation. Universitá Ca' Foscari Venezia.

[7] Pavle Subotić, Lazar Milikić, and Milan Stojić. 2022. A Static Analysis Framework for Data Science Notebooks. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 13–22. https://doi.org/10.1145/3510457.3513032

[8] Caterina Urban. 2020. What Programs Want: Automatic Inference of Input Data Specifications. *CoRR* abs/2007.10688 (2020). arXiv:2007.10688 https://arxiv.org/abs/2007.10688

[9] Caterina Urban and Antoine Miné. 2021. A Review of Formal Methods applied to Machine Learning. *ArXiv* abs/2104.02466 (2021). https://doi.org/10.48550/arXiv.2104.02466

[10] Caterina Urban and Peter Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 683–710. https://doi.org/10.1007/978-3-319-89884-1_24

[11] Ke Yang, Biao Huang, Julia Stoyanovich, and Sebastian Schelter. 2020. Fairness-Aware Instrumentation of Preprocessing Pipelines for Machine Learning, In Workshop on Human-In-the-Loop Data Analytics (HILDA'20). *Workshop on Human-In-the-Loop Data Analytics (HILDA'20)*. https://par.nsf.gov/biblio/10182459

Received 2023-03-10; accepted 2023-04-21