École Polytechnique - Bachelor of Science

Towards Verifying Data Science Software

Guruprerana Shabadi

Advised by Caterina Urban at

Inria & École Normale Supérieure — Université PSL

Abstract

Availability of data at extensive scales has led to the adoption of data-driven decision making practices in several domains including the likes of medicine, finance, and public policy. Consequently, ensuring correctness and fairness of analyses drawn from data science programs is critical and presents a challenging verification task. Errors in data processing code often go unnoticed since they do not throw errors, and can lead to unsound conclusions being drawn from data.

We propose an abstract interpretation based static analysis tool as a first step towards analysing data science programs. To this end, we construct a novel abstract domain to track data transformations which we exploit towards the task of shape inference of input data. Our tool is capable of automatically inferring the constraints and assumptions placed by a program on the input data. This can be further used towards achieving bigger objectives like input data usage analysis, data provenance, and the detection of bias or skew introduced in the data by a program.

Acknowledgements

The results presented in this thesis are part of a larger collaboration between the teams of Caterina Urban (Inria, Paris) and Pietro Ferrara (Università Ca' Foscari Venezia). During my time at Inria, I got to interact and work with a diverse and creative team. I am deeply grateful to my advisor Caterina Urban who entrusted me with this project at its early stages and guided me through the theoretical aspects of the analysis, especially since I came with no background in abstract interpretation. I would like to specially thank Luca Negrini (Università Ca' Foscari Venezia & Corvallis Srl) for patiently listening to all the iterations of my ideas to construct the dataframe graph abstract domain, helping me through Java programming for the implementation, and for being a wonderful friend at work. A special mention to everyone at the ANTIQUE team at ENS, Paris, who ensured that there was never a dull moment at work!

1 Introduction

The rocketing capabilities of today's computing systems to collect and process data at extensive scales has allowed data science and machine learning to permeate several domains where decisions are being driven by data. Ensuring correctness and fairness of data-driven analyses are particularly important when they are employed in safety-critical environments like medicine, public policy, or finance. Programming errors while processing and analysing data can have grave consequences leading us to draw incorrect conclusions. "Growth in a Time of Debt" was a paper published in 2010 by economists Reinhart and Rogoff which influenced the adoption of austerity policies for countries with various levels of public debt. However, the results presented in this paper were heavily criticised by experts [1]. In particular, a programming error was pointed out which led to the exclusion of a part of the data in the original analysis. Another recent example is that of Public Health England, where the size limit of documents in Microsoft's Excel software was the reason nearly 16,000 coronavirus cases went unreported, which meant that contact tracing could not be carried out for these cases[1].

In this thesis, we argue for the need to build verification tools for data science and data pre-processing programs. These programs commonly involve processing raw data by applying data transformations and deriving analyses about the data with the help of visualisation tools. While robustness and fairness verification of *trained* machine learning models like neural networks has become a popular research thrust [2], little attention has been given to the analysis of the data pre-processing stage within the machine learning pipeline. Data processing is also a tedious procedure because raw data is often inconsistent or incomplete. As a result, code written for data processing is seen as a one-off script that is rarely tested which makes it a very fragile part of the ML pipeline. Programming errors in this stage can go unnoticed because *they do not throw errors* and can be very difficult to debug in general. These errors can propagate into the training phases of the ML algorithms, resulting in below-par models [3].

The task of verification of data processing programs would be a composition of numerous smaller objectives. One of them is **input data usage inference** which involves statically deducing which subsets of the input data are *used* or remain *unused* by the program to produce a specific result. This is a crucial problem as demonstrated by the examples previously where it was not ensured that all the data was used. Equivalently, we could use the data usage information to check whether **senstitive data** was used while producing a result. For example, we often require analyses or predictions to be *independent* of sensitive features of the data like gender. Furthermore, we could also use this information to catch the **data leakage** bug in ML. Data leakage refers to the accidental sharing of information between the training and test datasets. As demonstrated by an example in [4], this bug is likely to occur within Jupyter notebook environments and often goes unnoticed. Yet another application of

[1] `https://www.bbc.com/news/technology-54423988`

the usage information can be towards automating **data provenance** tracking since data scientists often need to determine which parts of the input data was used to generate a result [5].

Detecting data transformations that introduce bias or skews is another important objective. As demonstrated in [6], data transformations like filter, concatenation, or missing value imputation can introduce bias and skew within the data. Concretely, consider the simple example of a dataset of a survey of a population, which contains details of the age and salary of each person. If a filter operation selects the rows with a salary above a certain number, then the operation potentially introduces a bias with respect to the age because there might be a statistical correlation between salary and age.

A final objective of our verification would be the **shape inference of input data**. While building a notebook, a data scientist writes the code with the structure of the input data in mind. Each transformation applied to the data assumes a certain structure on the *input* data. For example, a simple operation like a projection onto a column, assumes the *existence* of this column. By looking at the sequence of transformations used, it is possible to gather a variety of information about the shape of input data coming in from external files or other sources. The main motivation behind this is to help the data scientist understand the assumptions placed on the data by the code and check whether the code would generalise to other input data. For example, code that processes weekly COVID-19 data would need to work with potentially differently structured data.

In this work, we propose a static analysis tool based on abstract interpretation, that can help analyse Jupyter notebooks[2] which are popular environments used by data scientists to process and visualize raw data. Our analysis aims to achieve the last objective mentioned above: extracting constraints placed on input data by the program. This is an essential first step in the verification of data science programs before targeting the other objectives because we need as much information as possible about the input data before being able to study data usage or the introduction of bias.

Contributions and outline We build an abstract interpretation based static analysis tool for the shape inference of input data to programs. To achieve this, we propose a novel *dataframe graph* abstract domain which is capable of capturing the sequence of transformations applied to data. We begin by introducing the structure of data science notebooks in Section 2. In Section 3, we present the concrete semantics of data science programs which constitutes all the information that we would like to collect regarding the program. This is followed by Section 4 which defines the dataframe graph abstract domain. We also set up the link between the abstract and the concrete semantics through the concretization and prove its soundness. In Section 5, we discuss how we can use our abstract domain to help infer constraints placed on input data. Finally, we give some implementation details in Section 6.

[2] `https://jupyter.org/`

Related work Recently, obtaining formal guarantees on the safety and fairness of machine learning models has been a subject of widespread interest [2]. Our work builds upon this large line of work and proposes a verification framework for the data pre-processing stage of the machine learning pipeline. Our approach towards the shape inference of input data follows the work of [7] and extends it to support inputs to programs which contain datasets. Similarly, our objective of inferring input data usage directly follows from the compound data structure usage analysis presented in [8] and adds the ability to track the usage of selections of datasets. The closest body of work is probably [4] which also analyses data science notebooks along with their peculiar execution semantics and proposes an abstraction to detect data leakage.

The objective to detect bias/skew introduction is inspired from the `mlinspect` tool proposed in [9]. 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 tracking system for Python scripts. However, this tool requires executing the code which might not always be feasible when large datasets are involved.

1.1 Preliminaries

Order theory A *partial order* is a reflexive, transitive, and antisymmetric binary relation over a set. A set D endowed with a partial order relation $\sqsubseteq \subseteq D \times D$ is referred to as a *poset*. If we can further define the least upper bound (lub) $\sqcup$ and greatest lower bound (glb) $\sqcap$ operators on D respecting the partial order, then D is a *lattice* if for all $x, y \in D$, $x \sqcup y$ and $x \sqcap y$ are also elements of D. Finally, we say that D is a *complete lattice* if for each $X \sqsubseteq D$, $\sqcup X, \sqcap X \in D$. A complete lattice is then represented as $(D, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ where $\top$ and $\bot$ are the greatest and least elements of the lattice given by $\top \stackrel{\text{def}}{=} \sqcup D$ and $\bot \stackrel{\text{def}}{=} \sqcap D$ respectively.

Abstract interpretation Abstract interpretation [10, 11] is a powerful mathematical framework to soundly approximate the semantics of a program. The *concrete semantics* of a program refers to all the information that can be collected about the behaviour of programs at each point of execution and is an element of a complete lattice $(C, \sqsubseteq_C, \sqcup_C, \sqcap_C)$ known as the *concrete domain*. The *abstract semantics* which is an element of another complete lattice $(A, \sqsubseteq_A, \sqcup_A, \sqcap_A)$ is constructed to approximate the concrete semantics. The abstract semantics also carry the notion of being a decidable representation of the concrete semantics. The minimum relationship that we require between the abstract and the concrete worlds is a concretization function $\gamma : A \to C$ which preserves the partial order and the join operations of the lattices. The concrete semantics are represented as a fixpoint of a function $f : C \to C$ which

is often not computable. For this reason, we approximate f with the help of an abstract function $f^\sharp : A \to A$. $f^\sharp$ is a sound abstraction of f if for every $a \in A$, $f(\gamma(a)) \sqsubseteq_C \gamma(f^\sharp(a))$.

2 Data science notebooks

Data science notebooks, also known as Jupyter notebooks, are composed of a sequence of modifiable *cells*: these could be *code cells* which can contain a block of Python code and can be interleaved with visualisations and text to support the code. The code cells can be executed in any order at the user's request and the output of the execution, if any, is displayed below the cell. A previously executed cell can also be re-executed. The user is free to add, delete, and edit cells and execute them on demand. This provides the ability to rapidly prototype code and build incrementally which greatly increases the productivity of a data scientist. The code within data science notebooks typically follows a pattern:

- Notebooks usually begin by loading in data from a file or other sources.
- Several external libraries are used to transform and visualize the data like `pandas`, `numpy`, or `scikit-learn`.
- The processed data is then exported or used to train a machine learning model.

In the current implementation, we focus our efforts on building the analysis around one external library `pandas`. `pandas` is a popular framework which provides powerful tools to manipulate *dataframes*. Dataframes are the main objects of this library and can be thought of as a table with columns and multiple rows of data. There are several different functions that can be used to transform dataframes to produce new dataframes, each with multiple optional parameters. To avoid modeling each function implemented by `pandas`, we identified a small number of generic *dataframe transformation functions* which can capture the semantics of multiple `pandas` functions. A dataframe transformation function acts on one or more dataframes and produces a new dataframe as a result. A non-exhaustive list of these generic functions and their corresponding `pandas` functions are presented in Table 1.

Example 2.1 (Running example). We will use this small piece of example data science code in Python which works with COVID-19 data as an example to help explain our analysis intuitively.

```
1 import pandas as pd
2
3 df1 = pd.read_csv('covid-19-india.csv')
4 df1 = df1.loc['spo2' < 95]
5 df1['age'].replace(to_replace=['Under 20','Middle age','60+'],
      value=[1, 2, 3], inplace=True)
6
```

Transformation function	**Description**	**Corresponding `pandas` functions**
READ⟦`file`⟧	Initializes a dataframe by reading from a file	`pandas.read_csv(file)`
FILTER⟦`f`⟧	Selects the rows of the dataframe which satisfy the boolean function `f`	`df.loc[lambda row:f(row)]`
PROJECTION⟦`rows`, `cols`⟧	Project a dataframe along the specified rows and columns	`df.loc`, `df.iloc`, `df.drop`
TRANSFORM⟦`f`⟧	Apply the transformation function `f` along each row	`df.apply(f)`
ASSIGN⟦`selection`⟧	Assign another dataframe to a `selection` of a dataframe	`df.loc[selection] = df2`
CONCAT_ROWS, CONCAT_COLS	Concatenate one or more dataframes along the rows or columns	`df.merge`, `pd.concat`, `df.join`

Table 1: Most common dataframe transformation functions

```
df2 = pd.read_csv('covid-19-france.csv')
df2 = df2.iloc[5:1000]

df3 = pd.concat([df1, df2])
```

We can take this code and rewrite the transformations (in Program 1) with our list of generic dataframe transformatiom functions (see Table 1).

Program 1: Translation of the Python program listed in Example 2.1

[1] `df1` ≔ READ⟦`'covid-19-india.csv'`⟧ ;
[2] `df1` ≔ FILTER⟦`'spo2'` > 5⟧(`df1`) ;
[3] `df1` ≔ TRANSFORM⟦`age_transform`⟧(`df1`) ;

[4] `df2` ≔ READ⟦`'covid-19-france.csv'`⟧ ;
[5] `df2` ≔ PROJECTION⟦`rows` $= 5 : 1000$⟧(`df2`) ;

[6] `df3` ≔ CONCAT_ROWS(`df1`, `df2`)[7]

In addition, the arbitrary order of execution of cells within notebooks adds a level of complexity to the static analysis. To this end, we optionally allow the user of our static analysis tool to specify the order of execution of the code cells. Using this order of execution, we can combine the code cells and assemble them into one program that we can run our analysis on.

3 Data science program semantics

Intuitively, the concrete semantics of a program should capture all the possible information that we can gather about the values of variables as well as the inputs. In our case, when dealing with programs that read and manipulate data

from files, we would like to capture all the constraints placed on the dataframes within a program. In other words, we want our semantics to assemble a *set* of possible dataframes that each variable can represent at each point in the program such that the program execution is without errors. Of course, when there is inevitably an error that can occur, the set of dataframes would be empty.

With this intuition in mind, we first define $\mathbb{D}$ which is the set of all *dataframes* of any dimension taking values in

$$\mathbb{V} = \mathbb{S} \cup \mathbb{Z} \cup \mathbb{F} \cup \{\texttt{null}\}$$

which is the set of all strings ($\mathbb{S}$), integers ($\mathbb{Z}$), and floats ($\mathbb{F}$), along with a `null` value. A dataframe is simply a table that has rows numbered sequentially $0, 1, 2, \ldots$ and columns whose titles are strings. Let $\mathcal{D}$ be the set of all (dataframe) variables of a program. Then, an environment $\rho : \mathcal{D} \to \mathcal{P}(\mathbb{D})$ is a map which associates to each program variable $D \in \mathcal{D}$ a set of possible dataframes $\rho(D) \in \mathcal{P}(\mathbb{D})$ that the variable can represent. Lastly, we denote by $\mathcal{E}$ the set of all environments.

Next, we need to provide a structure to our data science programs. We will be representing these programs as control-flow graphs (CFGs) written as a triple $(\mathcal{L}, \mathcal{I}, E)$. The set $\mathcal{L}$ consists of the nodes of the graph which represent program labels and $\mathcal{I} \subseteq \mathcal{L}$ is the set of initial program labels. The edges E are a set of triples of the form (l_1, A, l_2) which connect different program labels (nodes) and are labelled with an action A. We have three types of actions which are supported: assignment of a dataframe variable $d \in \mathcal{D}$ to a dataframe expression (DE), the assumption of a boolean expression (BE), and a `nop` action that represents an action that does not evolve the program state. A grammar for the different actions is shown in Figure 2. Note that $\mathbb{S}$ represents the set of strings. These set of actions are sufficient to represent all the features of a programming language including loops and conditionals.

Within our programs, a dataframe expression DE, can be constructed either from existing dataframes or with a finite list of constant string or numerical values. They can also be read in from a data stream like a file. While manipulating dataframes, we allow for arbitrary functions that can take in a finite list of dataframes and produce a new one. These are represented by $\Psi_{\mathbb{D}}$ which is the set of all terminating computable functions $\psi_{\mathbb{D}} : \mathbb{D}^n \to \mathbb{D} \cup \{\texttt{error}\}$ for any $n \in \mathbb{N}$ that can produce a new dataframe from an existing dataframe or throw an error. These functions directly correspond to the generic transformation functions described earlier in Table 1. With a slight abuse of notation, we represent the function applied to a dataframe variable $d \in \mathcal{D}$ within our grammar in Figure 2. Similarly, Φ is the set of all computable and terminating functions $\phi : \mathbb{V}^n \to \mathbb{D} \cup \{\texttt{error}\}$ for any $n \in \mathbb{N}$, that can produce a dataframe from a constant set of values. Lastly, $\Psi_{\mathbb{B}}$ is the set of all computable and terminating functions $\phi_{\mathbb{B}} : \mathbb{D} \to \{\texttt{tt}, \texttt{ff}, \texttt{error}\}$ that produce a boolean from a dataframe. Some examples of this kind of functions include checking whether a specific column is present in a dataframe, if the values of the dataframe satisfy some constraints, or whether there are sufficient rows of data present.

Example 3.1 (Another running example). While it is not directly evident how we can capture conditionals and loops within a program CFG, we present an example program with a conditional and a loop which we will translate into a CFG (Program 2). $\psi^1_{\mathbb{B}}, \psi^2_{\mathbb{B}} \in \Psi_{\mathbb{B}}$ and $\psi^1_{\mathbb{D}}, \psi^2_{\mathbb{D}}, \psi^3_{\mathbb{D}} \in \Psi_{\mathbb{D}}$ are arbitrary functions. The corresponding CFG is shown in Figure 1.

Program 2: Program referenced in Example 3.1

```
[1]df1 ≔ READ('foo.csv') ;
[2]if ψ¹_𝔹(df1) then
|  [3]df1 ≔ ψ¹_𝔻(df1) ;
else
|  [4]df1 ≔ ψ²_𝔻(df1) ;
[5]df2 ≔ READ('boo.csv') ;
[6]while ψ²_𝔹(df2) do
|  [7]df1 ≔ ψ³_𝔻(df1)
end[8]
```

With the structure of our CFG setup, we have all the necessary ingredients to define the concrete semantics of our data science programs. The semantics of a dataframe expression is given by a function $\Delta_D[\![DE]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{D})$ that maps an environment to a set of possible dataframes that the dataframe expression can represent within that environment. These functions are defined for each dataframe expression in Figure 3. Similarly, we can associate a filtering semantics to each boolean expression that is a function $\text{FILTER}[\![BE]\!] : \mathcal{E} \to \mathcal{E}$ which maps an environment to a new environment which contains a subset of the original dataframes associated to each dataframe variable which satisfy the boolean expression. Finally, before defining the semantics of a program, we need to define the semantics of an action A which is yet another function $\Delta_A[\![A]\!] : \mathcal{E} \to \mathcal{E}$. These semantics are defined in Figure 4.

Finally, the semantics of a program $P = (\mathcal{L}, E)$ is given by a function $\Delta[\![P]\!] : \mathcal{L} \to \mathcal{P}(\mathcal{E})$ that maps each program label to the set of all environments that are possible when the program execution reaches that point. It can be defined as a least fixpoint in the complete lattice $(\mathcal{L} \to \mathcal{P}(\mathcal{E}), \dot{\subseteq}, \dot{\cup}, \dot{\cap}, \bot, \top)$ which is the pointwise lifting of the powerset lattice of the set of environments. The complete semantics is then given by

$$\Delta[\![P]\!] \stackrel{\text{def}}{=} \mathsf{lfp}\, \Theta \tag{1}$$

where $\Theta : (\mathcal{L} \to \mathcal{P}(\mathcal{E})) \to (\mathcal{L} \to \mathcal{P}(\mathcal{E}))$ is a function defined as

$$\Theta(m) \stackrel{\text{def}}{=} l \in \mathcal{L} \mapsto \bigcup_{(l', A, l) \in E} \Delta_A[\![A]\!](m(l')) \tag{2}$$

Theorem 3.1. *The least fixed point of the function Θ exists and is given by the limit of an increasing sequence $\mathsf{lfp}\, \Theta = \lim x_n$ in which $x_0 = \bot$ and $x_{n+1} = x_n \dot{\cup} \Theta(x_n)$ for all $n \geq 1$.*

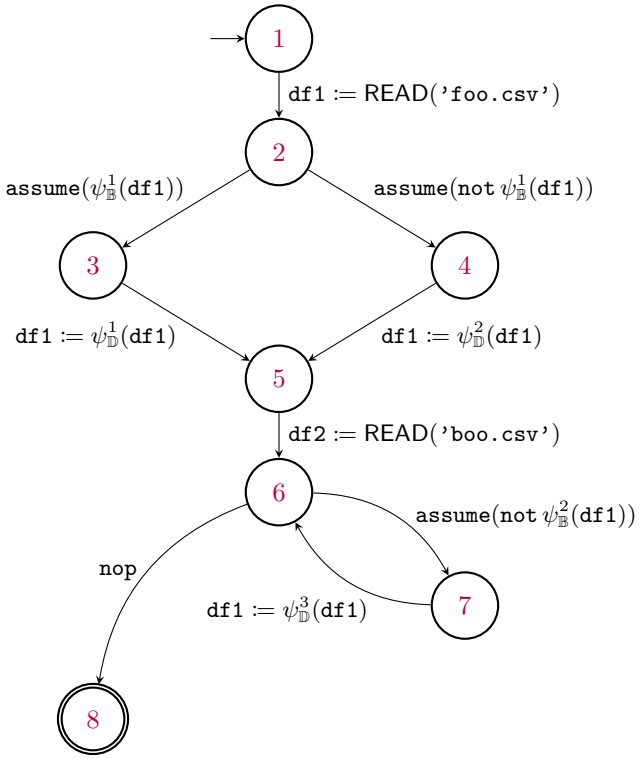


Figure 1: CFG of the data science program in Program 2

$$
\begin{array}{rcll}
BE & ::= & \psi_{\mathbb{B}}(d) & \psi_{\mathbb{B}} \in \Psi_{\mathbb{B}}, d \in \mathcal{D} \\
\\
DE & ::= & \phi(v_1, \ldots, v_n) & \phi \in \Phi, v_1, \ldots, v_n \in \mathbb{V} \\
 & | & \psi_{\mathbb{D}}(d_1, \ldots, d_n) & \psi_{\mathbb{D}} \in \Psi_{\mathbb{D}}, d_1, \ldots, d_n \in \mathcal{D} \\
 & | & \texttt{READ}(file) & file \in \mathbb{S} \\
\\
A & ::= & d \coloneqq DE \mid \texttt{assume}(BE) \mid \texttt{nop} & d \in \mathcal{D}
\end{array}
$$

Figure 2: Types of actions within the CFG

$$\Delta_D[\![\phi(v_1,\ldots,v_n)]\!] \stackrel{\text{def}}{=} \lambda\rho.\,(\{\phi(v_1,\ldots,v_n)\} \setminus \{\texttt{error}\}), \quad \phi \in \Phi, v_1,\ldots,v_n \in \mathbb{V}$$

$$\Delta_D[\![\psi_{\mathbb{D}}(d_1,\ldots,d_n)]\!] \stackrel{\text{def}}{=} \lambda\rho.\,\{\psi_{\mathbb{D}}(d_1^*,\ldots,d_n^*) \mid d_1^* \in \rho(d_1),\ldots,d_n^* \in \rho(d_n)\} \setminus \{\texttt{error}\} \quad \psi_{\mathbb{D}} \in \Psi_{\mathbb{D}}, d_1,\ldots,d_n \in \mathcal{D}$$

$$\Delta_D[\![\texttt{READ}(file)]\!] \stackrel{\text{def}}{=} \lambda\rho.\mathbb{D}, \quad file \in \mathbb{S}$$

Figure 3: Concrete semantics of dataframe expressions

$$\Delta_A[\![d := DE]\!] \stackrel{\text{def}}{=} \lambda\rho.\lambda x.\begin{cases}\Delta_D[\![DE]\!](\rho), & x = d\\ \rho(x), & x \in \mathcal{D} \setminus \{d\}\end{cases}$$
$$\Delta_A[\![\texttt{assume}(BE)]\!] \stackrel{\text{def}}{=} \lambda\rho.\text{FILTER}[\![BE]\!](\rho)$$
$$\Delta_A[\![\texttt{nop}]\!] \stackrel{\text{def}}{=} \lambda\rho.\rho$$

Figure 4: Concrete semantics of data science CFG actions

Proof. This result for the least fixed point is a direct consequence of Kleene's fixpoint theorem in the complete lattice $(\mathcal{L} \to \mathcal{P}(\mathcal{E}), \dot{\subseteq}, \dot{\cup}, \dot{\cap}, \bot, \top)$ and the continuity of Θ. □

Example 3.2. Reconsider the program in Example 2.1 with the translated transformations. Let us denote this program as P. The CFG form of this program would simply be a chain of nodes since there are no loops or conditionals. Computing the concrete semantics is fairly simple since the sequence to compute the least fixpoint would converge in 6 iterations (corresponding to the number of nodes in the CFG). In the end, we would obtain a set of environments containing the possible set of dataframes that each variable can represent at each program label, which is the complete concrete semantics of our program.

At the program label 3, we would have that

$$\Delta[\![P]\!](3) = \{\mathcal{E}_1\}$$

with

$$\mathcal{E}_1 = \begin{cases}\texttt{df1} & \mapsto \mathsf{TRANSFORM}(\mathsf{FILTER}(\mathbb{D}))\\ \texttt{df2} & \mapsto \phi\\ \texttt{df3} & \mapsto \phi\end{cases}$$

where we have one environment with the first variable mapping to the composition of the two transformations applied to it and the other two variables

mapping to the empty sets since they have not yet been defined. We have also omitted the transform and filter functions here for the sake of brevity. Note that we have only one environment since there are no loops or conditionals. Similarly at the program label 7, we would have

$$\Delta[\![P]\!](7) = \{\mathcal{E}_1\}$$

with

$$\mathcal{E}_1 = \begin{cases} \texttt{df1} & \mapsto \mathsf{TRANSFORM}(\mathsf{FILTER}(\mathbb{D})) \\ \texttt{df2} & \mapsto \mathsf{PROJECTION}(\mathbb{D}) \\ \texttt{df3} & \mapsto \mathsf{CONCAT\_ROWS}(\mathsf{TRANSFORM}(\mathsf{FILTER}(\mathbb{D})), \mathsf{PROJECTION}(\mathbb{D})) \end{cases}$$

4 Dataframe graph abstract domain

While the concrete semantics capture all possible information regarding the dataframes of our program, it is close to impossible to compute these in practice. Indeed, it is very hard to enumerate or store all the possible dataframes that a variable can represent. In search of a decidable representation of sets of dataframes, we construct a natural abstraction of the concrete semantics $\Delta[\![P]\!]$ of a data science program which is able to over-approximate the semantics at each program label $l \in \mathcal{L}$. In essence, we would like to build a representation of sets of dataframes that can help us infer properties of the structure and content of the dataframes. The key observation which will allow us to do this is that a dataframe's content is determined by its initial content and all the operations done on the dataframe. However, since dataframes are mainly initialized by reading from a file or a stream external to the program, so we are unable to capture any information about the initial data. On the other hand, we are able to track all the operations done on the dataframe which *can* give us a lot of information about the dataframe. We build this abstraction with the help of graphs which can track the sequence of operations done on a dataframe.

Employing the theory of abstract interpretation, we derive the abstraction $\Delta^\sharp[\![P]\!] : \mathcal{L} \to \mathcal{E}^\sharp$, where $\mathcal{E}^\sharp$ is the set of abstract environments. An abstract environment is a function $\rho^\sharp : \mathcal{D} \to \mathbb{T}$ in which $\mathbb{T}$ is our *dataframe graph abstract domain*. While building the abstraction, we require that the abstract environment at a label $l \in \mathcal{L}$ given by $\Delta^\sharp[\![P]\!](l)$ is an over-approximation of the set of environments at l given by the concrete semantics $\Delta[\![P]\!](l)$.

4.1 Dataframe graphs

We first describe the construction of our dataframe graph domain $\mathbb{T}$. It is the set of directed graphs of the form $(\mathcal{O}, E, \mathsf{Init}, \mathsf{Exit})$ where $\mathcal{O}$ is the set of nodes of the graph. Each node is associated with an operation within the set $\mathbb{O}$ defined by

$$\mathbb{O} \stackrel{\text{def}}{=} \Psi_{\mathbb{D}} \cup \Psi_{\mathbb{B}} \cup \{\phi(v_1, \ldots, v_n) \mid \phi \in \Phi, v_1, \ldots, v_n \in \mathbb{V}, n \in \mathbb{N}\} \cup \{\texttt{READ}(file) \mid file \in \mathbb{S}\} \cup \{\texttt{OR}\} \quad (3)$$

These possible operations correspond to dataframe operations given by the dataframe and boolean expressions within the actions of our program (Figure 2), along with an additional operation `OR`. Thus, the nodes of a dataframe graph $\mathcal{O}$ is a subset $\mathcal{O} \subseteq \{(l, o) \mid l \in \mathbb{N}, o \in \mathbb{O}\}$ where the first component is the label of the node. $E \subseteq \{(o_1, n, o_2) \mid o_1, o_2 \in \mathcal{O}, n \in \mathbb{N}\}$ is the set of directed edges within the graph which is also annotated with a number $n \in \mathbb{N}$ which serves to track the order of edges. $\mathsf{Init} \subseteq \mathcal{O}$ is the set of entry points of the graph and $\mathsf{Exit} \in \mathcal{O}$ is the unique terminal node of the graph.

Before moving forward, we define some important notation required to work with dataframe graphs. For each node $o \in \mathcal{O}$ of a graph, we denote by $\mathsf{pred}(o)$ the tuple of predecessors of o ordered by the annotated number on the edges leading to o. $\mathsf{npred}(o)$ is the number of predecessors of the node and $\mathsf{pred}(o, i)$ denotes the i-th predecessor of the node which is the i-th element of $\mathsf{pred}(o)$. We also denote by $\mathsf{succ}(o)$ the set of successor nodes of o. $\mathsf{op}(o) \in \mathbb{O}$ denotes the operation associated with o. Lastly, $\mathsf{depth}(o)$ denotes the length of the shortest directed path from o to the Exit node. We also place some additional restrictions on a graph $(\mathcal{O}, E, \mathsf{Init}, \mathsf{Exit}) \in \mathcal{T}$ within our dataframe graph abstract domain:

1. The entry nodes of the graph

$$\mathsf{Init} \subseteq \{\phi(v_1, \ldots, v_n) \mid \phi \in \Phi, v_1, \ldots, v_n \in \mathbb{V}, n \in \mathbb{N}\} \cup \{\texttt{READ}(file) \mid file \in \mathbb{S}\},$$

   which means that they are required to be represented by an initialization operation either read from a file or constructed from a set of constant values. They also do not have any predecessors.

2. All the edges leading to a node $o \in \mathcal{O}$ can be uniquely ordered based on the annotated numbers on the edges.

3. There is a path from every node to the Exit node.

4. Each node $o \in \mathcal{O}$ with a boolean operation $\mathsf{op}(o) \in \Psi_{\mathbb{B}}$ satisfies $\mathsf{npred}(o) = 1$, i.e., has exactly only one predecessor.

Further restrictions can be placed on the graphs similar to [12] like restrictions on cycles and sharing on nodes between subgraphs. Rather than place these restrictions on our domain, we will be able to deduce them as *properties* of our dataframe graphs generated by the abstract semantics of our program.

Equipped with our definition of the set of dataframe graphs $\mathbb{T}$, we define lattice operations on these graphs to obtain a complete lattice $(\mathbb{T}, \sqsubseteq, \sqcup, \top, \bot)$.

Example 4.1. Dataframe graphs help us track the operations done on dataframes. Considering again the program in Example 2.1, we expect the abstract semantics $\Delta^\sharp[\![P]\!]$ to contain a graph associated to each variable at every program label. At the final program label 7, the graph associated to `df3` would be the the graph shown in Figure 5 given by $\Delta^\sharp[\![P]\!](7)(\texttt{df3})$. The numbers on the edges demarcate the order on the edges with respect to the destination node. For example, there are two incoming edges to the concatenation node at the end since the order of concatenation of the dataframes is important.

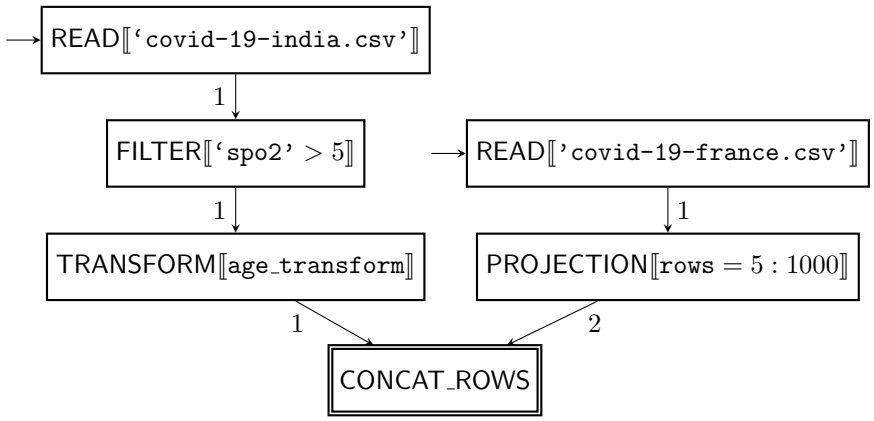


Figure 5: Graph of `df3` at the program label 7 of Example 2.1

Concretization Each dataframe graph in $\mathbb{T}$ is an overapproximation of a set of sets of dataframes. To interact with the concrete world, we define a concretization function $\gamma : \mathbb{T} \to \mathcal{P}(\mathcal{P}(\mathbb{D}))$ which gives the set of set of dataframes that a given graph represents. Again, we define this function recursively starting from the exit nodes. Let $T = (\mathcal{O}, E, \mathsf{Init}, \mathsf{Exit}) \in \mathbb{T}$ be a dataframe graph. We define

$$\gamma(T) \stackrel{\text{def}}{=} \begin{cases} \{\phi\}, & T = \bot \\ \{\mathcal{D}\}, & T = \top \\ \gamma_o(\mathsf{Exit}), & \text{otherwise} \end{cases} \tag{4}$$

where we define another function $\gamma_o : \mathcal{O} \to \mathcal{P}(\mathcal{P}(\mathbb{D}))$

$$\gamma_o(o) \stackrel{\text{def}}{=} \begin{cases} \{\mathbb{D}\}, \quad \mathsf{op}(o) = \texttt{READ}(file), file \in \mathbb{S} \\ \{\{\phi(v_1, \ldots, v_n)\} \setminus \{\texttt{error}\}\}, \\ \qquad \mathsf{op}(o) = \phi(v_1, \ldots, v_n), \phi \in \Phi, v_1, \ldots, v_n \in \mathbb{V}, n \in \mathbb{N} \\ \bigcup_{\substack{s_i \in \gamma_o(\mathsf{pred}(o,i)) \\ 1 \le i \le \mathsf{npred}(o)}} \{\psi_{\mathbb{D}}(s_1, \ldots, s_{\mathsf{npred}(o)}) \setminus \{\texttt{error}\}\}, \\ \qquad\qquad \mathsf{op}(o) = \psi_{\mathbb{D}} \in \Psi_{\mathbb{D}} \\ \{\mathrm{FILTER}[\![\psi_{\mathbb{B}}]\!](s) \mid s \in \gamma_o(\mathsf{pred}(o,1))\}, \quad \mathsf{op}(o) = \psi_{\mathbb{B}} \in \Psi_{\mathbb{B}} \\ \bigcup_{1 \le i \le \mathsf{npred}(o)} \gamma_o(\mathsf{pred}(o,i)), \quad \mathsf{op}(o) = \texttt{OR} \end{cases} \tag{5}$$

γ_o recursively evaluates the sets of dataframes coming in from each of the incoming edges and then applies the function at the given node to these sets of dataframes. On the other hand, when we have a `OR` node, we simply take a union of all the sets of dataframes coming in from each of the predecessors. With this, we are able to extend the definition of γ to an abstract environment

$\rho^\sharp : \mathcal{D} \to \mathbb{T}$ to obtain the set of concrete environments that it approximates

$$\gamma(\rho^\sharp) \stackrel{\text{def}}{=} \left\{\rho \in \mathcal{E} \mid \forall d \in \mathcal{D}, \rho(d) \in \gamma(\rho^\sharp(d))\right\} \tag{6}$$

Example 4.2. In Figure 5, we showed that the graph of `df3` at the program label 7 of the program in Example 2.1 given by the abstract semantics. If we evaluate the concretization function on this graph, we would obtain

$$\gamma(\Delta^\sharp[\![P]\!](7)(\texttt{df3})) = \\ \{\mathsf{CONCAT\_ROWS}(\mathsf{TRANSFORM}(\mathsf{FILTER}(\mathbb{D})), \mathsf{PROJECTION}(\mathbb{D}))\}$$

which corresponds to the set of values given by the concrete semantics as determined in Example 3.2. The concretization function iterates through the graph starting from the exit node and composes the functions on the nodes.

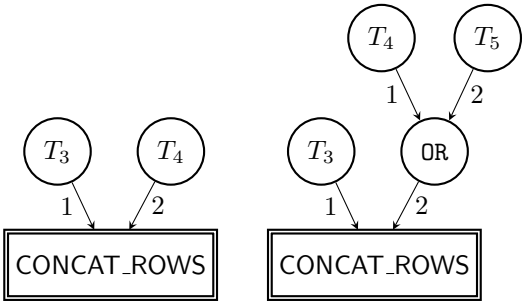
Partial order The partial order on $\mathbb{T}$ which we denote by $\sqsubseteq$ is defined by the Algorithm 1 which can compute for any $T_1, T_2 \in \mathbb{T}$,

$$T_1 \sqsubseteq T_2 \stackrel{\text{def}}{=} \begin{cases} \texttt{tt}, & T_2 = \top \\ \texttt{tt}, & T_1 = \bot \\ \texttt{ff}, & T_1 = \top \\ \texttt{ff}, & T_2 = \bot \\ \mathsf{leq}(\mathsf{Exit}_1, \mathsf{Exit}_2, \phi), & \text{otherwise} \end{cases}$$

where Exit_1 and Exit_2 are the exit nodes of T_1 and T_2 respectively, and $\texttt{tt}, \texttt{ff}$ represent the standard boolean values. We define the leq function recursively which traverses both the graphs iteratively starting from the exit nodes of the graphs and then going through the predecessors.

Algorithm 1: $\mathsf{leq}(o_1, o_2, M)$

if $(o_1, o_2) \in M$ **then**
  ret `tt`
else if $\mathit{op}(o_1) = \mathit{op}(o_2) \neq \mathit{OR}$ **then**
  if $\mathit{npred}(o_1) = \mathit{npred}(o_2)$ **then**
    ret $\forall 1 \leq i \leq \mathsf{npred}(o_1), \mathsf{leq}\,(\mathsf{pred}(o_1, i), \mathsf{pred}(o_2, i), M \cup \{(o_1, o_2)\})$
else if $\mathit{op}(o_1) = \mathit{op}(o_2) = \mathit{OR}$ **then**
  ret $\forall 1 \leq i_1 \leq \mathsf{npred}(o_1), \exists 1 \leq i_2 \leq \mathsf{npred}(o_2),$
    $\mathsf{leq}\,(\mathsf{pred}(o_1, i_1), \mathsf{pred}(o_2, i_2), M \cup \{(o_1, o_2)\})$
else if $\mathit{op}(o_1) = \mathit{OR}, \mathit{op}(o_2) \neq \mathit{OR}$ **then**
  ret $\forall 1 \leq i \leq \mathsf{npred}(o_1), \mathsf{leq}\,(\mathsf{pred}(o_1, i), o_2, M \cup \{(o_1, o_2)\})$
else if $\mathit{op}(o_1) \neq \mathit{OR}, \mathit{op}(o_2) = \mathit{OR}$ **then**
  ret $\exists 1 \leq i \leq \mathsf{npred}(o_2), \mathsf{leq}\,(o_1, \mathsf{pred}(o_2, i), M \cup \{(o_1, o_2)\})$
else
  ret `ff`



(a) The graph T_1 (b) The graph T_2

Figure 6: The graphs used in Example 4.3

Theorem 4.1 (Soundness of the partial order)**.** *The partial order defined on* $\mathbb{T}$ *is sound with respect to the defined concretization function* $\gamma : \mathbb{T} \to \mathcal{P}(\mathcal{P}(\mathbb{D}))$*, i.e., for all* $T_1, T_2 \in \mathbb{T}$*, if* $T_1 \sqsubseteq T_2$*, then* $\gamma(T_1) \subseteq \gamma(T_2)$*.*

Proof (sketch). Since both the partial order and concretization are defined recursively, we can prove this with the help of an induction over the structure of the graphs.

This is straightforward because the algorithm to determine the partial order between two graphs is constructed to mimic what happens in the concrete world through the concretization function. Since each graph represents a set of sets of dataframes, we want the natural inclusion to be satisfied when the order relation is satisfied. □

Example 4.3. Two dataframe graphs $T_1, T_2 \in \mathbb{T}$ are shown in Figure 6. Suppose $T_3, T_4, T_5 \in \mathbb{T}$ are other graphs which form subgraphs in T_1 and T_2. The concretization of T_1 gives us

$$\gamma(T_1) = \{\mathsf{CONCAT\_ROWS}(s_1, s_2) \mid s_1 \in \gamma(T_3), s_2 \in \gamma(T_4)\}$$

and that of T_2,

$$\begin{aligned}\gamma(T_2) = \{&\mathsf{CONCAT\_ROWS}(s_1, s_2) \mid s_1 \in \gamma(T_3), s_2 \in \gamma(T_4)\} \\ &\cup \{\mathsf{CONCAT\_ROWS}(s_1, s_2) \mid s_1 \in \gamma(T_3), s_2 \in \gamma(T_5)\}\end{aligned}$$

As $\gamma(T_1) \subseteq \gamma(T_2)$, we would expect $T_1 \sqsubseteq T_2$ which is indeed the case as determined by the partial order algorithm, which handles the `OR` node appropriately.

Least upper bound We define the least upper operator on $\mathbb{T}$ denoted by $\sqcup$ as follows

$$\underbrace{(\mathcal{O}_1, E_1, \mathsf{Init}_1, \mathsf{Exit}_1)}_{T_1} \sqcup \underbrace{(\mathcal{O}_2, E_2, \mathsf{Init}_2, \mathsf{Exit}_2)}_{T_2} \stackrel{\text{def}}{=} \begin{cases} T_1, & T_1 \sqsubseteq T_2 \\ T_2, & T_2 \sqsubseteq T_1 \\ (\mathcal{O}^{\sqcup}, E^{\sqcup}, \mathsf{Init}^{\sqcup}, \mathsf{Exit}^{\sqcup}), & \text{otherwise} \end{cases} \tag{7}$$

where

$$\begin{aligned} \mathcal{O}^{\sqcup} &\stackrel{\text{def}}{=} \mathcal{O}_1 \cup \mathcal{O}_2 \cup \{(\texttt{freshlab}, \texttt{OR})\} \\ E^{\sqcup} &\stackrel{\text{def}}{=} E_1 \cup E_2 \cup \{(\mathsf{Exit}_1, 1, \texttt{freshlab}), (\mathsf{Exit}_2, 2, \texttt{freshlab})\} \\ \mathsf{Init}^{\sqcup} &\stackrel{\text{def}}{=} \mathsf{Init}_1 \cup \mathsf{Init}_2 \\ \mathsf{Exit}^{\sqcup} &\stackrel{\text{def}}{=} (\texttt{freshlab}, \texttt{OR}) \end{aligned}$$

Here, `freshlab` is simply a new label for a node which is not shared with any other node of the graph.

Theorem 4.2 (Soundness of the least upper bound operator)**.** *The least upper bound operator $\sqcup$ is sound with respect to the concretization function γ, i.e., for all $T_1, T_2 \in \mathbb{T}$, $\gamma(T_1) \cup \gamma(T_2) \subseteq \gamma(T_1 \sqcup T_2)$*

Proof (sketch). Since we only add a `OR` node after combining the graphs of T_1 and T_2, by the definition of the concretization applied to the exit `OR` node, we have that $\gamma(T_1 \sqcup T_2) = \gamma(T_1) \cup \gamma(T_2)$. □

Similar to the partial order, the least upper bound between two graphs is made to mimic what happens in the concrete world. The least upper bound in the concrete world of sets of sets of dataframes is given by the simple set union. Our least upper bound operator combines two graphs with an `OR` node which has the semantics of the set union as defined by the concretization function.

Example 4.4. As we will define soon, the least upper bound is used when we need to combine the semantics coming in from two or more branches of the CFG. An example of this is the program label 5 in the CFG of Figure 1 where the executions coming in from the conditional branches meet. While evaluating the abstract semantics, we would a graph representing `df1` coming in from the left branch and similarly for the right branch. Combining these at 5 using the least upper bound would give us the graph shown in Figure 7.

Widening Since the dataframe graph domain does not satisfy the ascending chain property (one can always add multiple `OR` nodes to get *bigger* graphs with respect to the defined partial order), we need to endow our graph domain with a widening operator ∇. However, since data science code is mostly consists of a

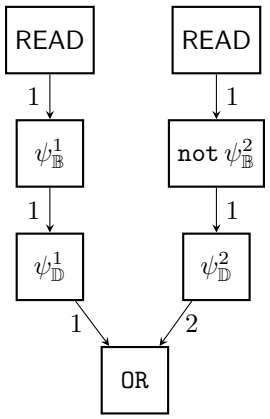


Figure 7: Graph of `df1` at program label 5 of the CFG in Figure 1 representing the least upper bound of the two graphs coming in from the two branches

sequential chain of transformations without loops, we choose to adopt the naive widening given by

$$\forall T_1, T_2 \in \mathbb{T}, \quad T_1 \nabla T_2 \stackrel{\text{def}}{=} \begin{cases} T_1, & T_2 \sqsubseteq T_1 \\ T_2, & T_1 \sqsubseteq T_2 \\ \top, & \text{otherwise} \end{cases} \tag{8}$$

4.2 Abstract semantics of data-science programs

Similar to how we defined the concrete semantics, we need to define the abstract semantics of each different type of action within our data-science program CFG. The abstract semantics of a dataframe expression (DE) is a function $\Delta^\sharp_D[\![DE]\!] : \mathcal{E}^\sharp \to \mathbb{T}$ which evaluates the dataframe expression within an environment and produces a new dataframe graph which is the abstract value of the dataframe expression. In the same vein, $\Delta^\sharp_B[\![BE]\!] : \mathcal{E}^\sharp \to \mathbb{T}$ evaluates a boolean expression and produces a new graph since we would also like to track boolean conditions within our dataframe graphs.

Abstract semantics of dataframe expressions Here we define the functions $\Delta^\sharp_D[\![DE]\!] : \mathcal{E}^\sharp \to \mathbb{T}$ for each dataframe expression (DE).

1. When constructing a new dataframe from a set of values, we just produce a new graph with one node and no edges.

$$\Delta^\sharp_D[\![\phi(v_1, \ldots, v_n)]\!] \stackrel{\text{def}}{=} \lambda\rho^\sharp.\,(\{o\}, \phi, \{o\}, o)\,, \quad o \stackrel{\text{def}}{=} (\texttt{freshlab}, \phi(v_1, \ldots, v_n)), \\ \phi \in \Phi, v_1, \ldots, v_n \in \mathbb{V}, n \in \mathbb{N} \tag{9}$$

2. Again, when reading from a file, we simply have a new graph with one node.

$$\Delta_D^\sharp[\![\texttt{READ}(file)]\!] \stackrel{\text{def}}{=} \lambda\rho^\sharp.\,(\{o\},\phi,\{o\},o)\,, \quad o \stackrel{\text{def}}{=} (\texttt{freshlab},\texttt{READ}(file)), \quad file \in \mathbb{S} \tag{10}$$

3. When constructing a new dataframe from one or more existing dataframes, we construct a new graph by combining the graphs of the dataframes and then add a node for the operation that acts on these previous graphs with incoming edges from the other graphs.

$$\begin{aligned}
\Delta_D^\sharp[\![\psi_{\mathbb{D}}(d_1,\ldots,d_n)]\!] &\stackrel{\text{def}}{=} \lambda\rho^\sharp.(\mathcal{O}^*,E^*,\mathsf{Init}^*,o), \\
&\qquad \psi_{\mathbb{D}} \in \Psi_{\mathbb{D}}, d_1,\ldots,d_n \in \mathcal{D} \\
\mathcal{O}^* &\stackrel{\text{def}}{=} \bigcup_{1\le i\le n} \mathcal{O}_i \cup \{o\} \\
E^* &\stackrel{\text{def}}{=} \bigcup_{1\le i\le n} E_i \cup \{(\mathsf{Exit}_i,i,o) \mid 1\le i\le n\} \\
\mathsf{Init}^* &\stackrel{\text{def}}{=} \bigcup_{1\le i\le n} \mathsf{Init}_i \\
o &\stackrel{\text{def}}{=} (\texttt{freshlab},\psi_{\mathbb{D}})
\end{aligned}$$

   where for all $1 \le i \le n$, we set $(\mathcal{O}_i, E_i, \mathsf{Init}_i, \mathsf{Exit}_i) \stackrel{\text{def}}{=} \rho^\sharp(d_i)$. An example of this is seen in Figure 5 where the graph of `df3` is constructed by combining the graphs of `df1` and `df2` adding a concatenation node with appropriately numbered edges to the new node (see Example 2.1).

Abstract semantics of boolean expressions We define the function $\Delta_B^\sharp[\![BE]\!] : \mathcal{E}^\sharp \to \mathbb{T}$ for boolean expressions (BE).

$$\begin{aligned}
\Delta_B^\sharp[\![\psi_{\mathbb{B}}(d)]\!] &\stackrel{\text{def}}{=} \lambda\rho^\sharp.(\mathcal{O}\cup\{o\},E\cup\{(\mathsf{Exit},1,o)\},\mathsf{Init},o), \quad \psi_{\mathbb{B}} \in \Psi_{\mathbb{B}}, d \in \mathcal{D} \\
(\mathcal{O},E,\mathsf{Init},\mathsf{Exit}) &\stackrel{\text{def}}{=} \rho^\sharp(d) \\
o &\stackrel{\text{def}}{=} (\texttt{freshlab},\psi_{\mathbb{B}})
\end{aligned}$$

Abstract semantics of the CFG actions Lastly, we need to define the functions $\Delta_A^\sharp[\![A]\!] : \mathcal{E}^\sharp \to \mathcal{E}^\sharp$ which are similar to the concrete semantics of actions (see Figure 8).

Abstract semantics of data-science programs Combining the abstract semantics of the actions of the CFG, we are able to define the abstract semantics of the complete CFG which is a function $\Delta^\sharp[\![P]\!] : \mathcal{L} \to \mathcal{E}^\sharp$ given by

$$\Delta^\sharp[\![P]\!] \stackrel{\text{def}}{=} \mathsf{lfp}^\sharp\,\Theta^\sharp \tag{11}$$

$$\Delta_A^\sharp[\![d \coloneqq DE]\!] \stackrel{\text{def}}{=} \lambda\rho^\sharp.\lambda x. \begin{cases} \Delta_D^\sharp[\![DE]\!](\rho^\sharp), & x = d \\ \rho^\sharp(x), & x \in \mathcal{D} \setminus \{d\} \end{cases}$$

$$\Delta_A^\sharp[\![\texttt{assume}(\psi_{\mathbb{B}}(d))]\!] \stackrel{\text{def}}{=} \lambda\rho^\sharp.\lambda x. \begin{cases} \Delta_B^\sharp[\![\psi_{\mathbb{B}}(d)]\!](\rho^\sharp), & x = d \\ \rho^\sharp(x), & x \in \mathcal{D} \setminus \{d\} \end{cases}, \quad \psi_{\mathbb{B}} \in \Psi_{\mathbb{B}}$$

$$\Delta_A^\sharp[\![\texttt{nop}]\!] \stackrel{\text{def}}{=} \lambda\rho^\sharp.\rho^\sharp$$

Figure 8: Abstract semantics of CFG actions

where $\Theta : (\mathcal{L} \to \mathcal{E}^\sharp) \to (\mathcal{L} \to \mathcal{E}^\sharp)$ is a function defined as

$$\Theta^\sharp(m) \stackrel{\text{def}}{=} l \in \mathcal{L} \mapsto \dot{\bigsqcup_{(l',A,l)\in E}} \Delta_A[\![A]\!](m(l')) \tag{12}$$

Note that we implicitly define a lattice structure on $\mathcal{L} \to \mathcal{E}^\sharp = \mathcal{L} \to (\mathcal{D}^\sharp \to \mathbb{T})$ through a pointwise lifting of the lattice $(\mathbb{T}, \sqsubseteq, \sqcup, \top, \bot)$. We also modify the computation of the the least fixpoint to ensure finite convergence by applying widening.

Definition 4.1. Let $N \in \mathbb{N}$ be a well-chosen parameter for a program. We define $\mathsf{lfp}^\sharp\, \Theta^\sharp \stackrel{\text{def}}{=} \lim x_n^\sharp$ where

$$\begin{aligned} x_0 &= \bot \\ x_{n+1} &= x_n \,\dot{\sqcup}\, \Theta^\sharp(x_n), \quad \forall n \leq N \\ x_{n+1} &= x_n \,\dot{\nabla}\, \Theta^\sharp(x_n), \quad \forall n > N \end{aligned}$$

Remark. The defined sequence to compute $\mathsf{lfp}^\sharp\, \Theta^\sharp$ converges in finite time as a consequence of applying the widening.

4.3 Soundness of the abstract semantics

So far we have constructed a sound abstraction of sets of sets of dataframes which we can succinctly represent by

$$(\mathcal{P}(\mathcal{P}(\mathbb{D})), \subseteq) \stackrel{\gamma}{\leftarrow} (\mathbb{T}, \sqsubseteq) \tag{13}$$

Then by extending the definition of the concretization function γ (by Equation 6) to abstract environments, we have the following abstraction of sets of environments

$$(\mathcal{P}(\mathcal{E}), \subseteq) \stackrel{\gamma}{\leftarrow} (\mathcal{E}^\sharp, \dot{\sqsubseteq}) \tag{14}$$

We can finally lift this abstraction pointwise with respect to the set of program labels $\mathcal{L}$ of a data-science program we obtain the following sound abstraction

$$(\mathcal{L} \to \mathcal{P}(\mathcal{E}), \dot{\subseteq}) \stackrel{\dot{\gamma}}{\leftarrow} (\mathcal{L} \to \mathcal{E}^\sharp, \dot{\sqsubseteq}) \tag{15}$$

Thus, for a data-science program P, in order for the abstract semantics $\Delta^\sharp[\![P]\!] : \mathcal{L} \to \mathcal{E}^\sharp$ to be an overapproximation of our concrete semantics $\Delta[\![P]\!] : \mathcal{L} \to \mathcal{P}(\mathcal{E})$, we require that

$$\Delta[\![P]\!] \dot{\subseteq} \dot{\gamma}(\Delta^\sharp[\![P]\!]) \tag{16}$$

Essentially, for each program label $l \in \mathcal{L}$, we need that $\Delta[\![P]\!](l) \subseteq \gamma(\Delta^\sharp[\![P]\!](l))$.

Theorem 4.3. *The abstract semantics of a data-science program $P = (\mathcal{L}, \mathcal{I}, E)$, $\Delta^\sharp[\![P]\!]$ is a sound over-approximation of the concrete semantics $\Delta[\![P]\!]$ as given by 16.*

Proof. Remember that from Theorem 3.1 we can write the concrete semantics of the program as the limit of a sequence of a set of functions $(x_n)_{n\in\mathbb{N}}$ in $\mathcal{L} \to \mathcal{P}(\mathcal{E})$ defined as

$$\begin{aligned} x_0 &= \bot \\ x_{n+1} &= x_n \dot{\cup} \Theta(x_n) \end{aligned}$$

Similarly, the abstract semantics is the limit of the sequence $(x_n^\sharp)_{n\in\mathbb{N}}$ in $\mathcal{L} \to \mathcal{E}^\sharp$ given by

$$\begin{aligned} x_0^\sharp &= \bot^\sharp \\ x_{n+1}^\sharp &= x_n \dot{\sqcup} \Theta^\sharp(x_n), \quad \forall n \leq N \\ x_{n+1}^\sharp &= x_n \dot{\nabla} \Theta^\sharp(x_n), \quad \forall n > N \end{aligned}$$

where $N \in \mathbb{N}$ is a predefined constant.

We claim that for each $n \in \mathbb{N}$, $x_n \dot{\subseteq} \dot{\gamma}(x_n^\sharp)$. Indeed, this is easily seen by an induction, assuming that $\Theta^\sharp$ is a sound approximation of Θ which we show in Lemma 4.3.1. Firstly, $x_0 = \bot \dot{\subseteq} \dot{\gamma}(x_0^\sharp) = \dot{\gamma}(\bot^\sharp)$. Now let $n \in \mathbb{N}$ and assume $x_n \dot{\subseteq} \dot{\gamma}(x_n^\sharp)$. If $n \leq N$, then by the soundness of the join operator and $\Theta^\sharp$, we obtain that

$$x_{n+1} = x_n \dot{\cup} \Theta(x_n) \dot{\subseteq} \dot{\gamma}(x_n^\sharp) \dot{\cup} \dot{\gamma}(\Theta^\sharp(x_n^\sharp)) \dot{\subseteq} \dot{\gamma}(x_n^\sharp \dot{\sqcup} \Theta^\sharp(x_n^\sharp))$$

For $n \geq N$, we treat it symmetrically using the soundness of the widening operator with respect to the concretization. Equipped with the result that for each $n \in \mathbb{N}$, $x_n \dot{\subseteq} \dot{\gamma}(x_n^\sharp)$, we can deduce

$$\Delta^\sharp[\![P]\!] = \lim x_n \dot{\subseteq} \dot{\gamma}\left(\lim x_n^\sharp\right) = \dot{\gamma}(\Theta^\sharp[\![P]\!])$$

where we pass to the limit implicitly assuming the continuity of $\dot{\gamma}$. □

Lemma 4.3.1. *Let $P = (\mathcal{L}, \mathcal{I}, E)$ be a data-science program. $\Theta^\sharp$ is a sound over-approximation of Θ, i.e., for all $m : \mathcal{L} \to \mathcal{E}^\sharp$, $\Theta(\dot{\gamma}(m)) \dot{\subseteq} \dot{\gamma}(\Theta^\sharp(m))$.*

Proof (sketch). Let $m : \mathcal{L} \to \mathcal{E}^\sharp$ and let $l \in \mathcal{L}$. We have that

$$\Theta(\dot{\gamma}(m))(l) = \bigcup_{(l',A,l)\in E} \Delta_A[\![A]\!](\dot{\gamma}(m)(l'))$$

and also that

$$\Theta^\sharp(m) = \dot{\bigsqcup}_{(l',A,l)\in E} \Delta_A^\sharp[\![A]\!](m(l'))$$

Utilizing the soundness of the abstract least upper bound operator $\dot{\sqcup}$, the task reduces to proving the soundness of the abstract action semantics functions $\Delta_A^\sharp[\![A]\!]$ with respect to the concrete action semantics $\Delta_A[\![A]\!]$ for each action A. The key observation here is that the abstract graph domain does not interpret the semantics of each of the dataframe or boolean expressions and only stores the function calls themselves. Therefore the abstract action semantics are sound by construction. □

5 Shape inference of input data

The dataframe graph abstraction provides the ideal structure to extract the assumptions and constraints placed on input data by the data science program. The key intuition behind this idea is that each dataframe transformation represents a *constraint* on the types of dataframes that it can transform. Propagating these constraints backwards through the dataframe graphs, we can obtain all the constraints placed by the transformations on the input data such that the execution of the program remains safe.

After computing the abstract semantics of a program and thereby the graphs representing all the variables, we can implement an analysis which explores these graphs and gathers a specific type of information about the input data files. In contrast with the dataframe graph abstraction which did not interpret the semantics of the transformation functions themselves, in order to collect constraints on the input data, the analysis requires knowledge about the constraints that can be inferred from each transformation. Here, having a small set of representative transformations can come in handy. Some analyses that can implemented include: inferring number of rows/columns, types of the data in each column (string or numerical), properties of the values of a subset of rows/columns, etc.

Example 5.1. Looking again at the program in Example 2.1, we can extract the following information:

- From line 3, we can infer that the input file has a column named `‘spo2’` and starting from that line, the values in this column will be less than 95 in `df1`.
- In line 5, we see that the `‘age’` column in the input is composed of strings possibly with values in `[‘Under 20’, ‘Middle age’, ‘60+’]`.

- From line 8, we know that the second input file has at least 1000 rows.

Inferring information about the input data is the first crucial step that can help us tackle the tougher challenges in analysing data science code. A first goal could be to **identify inconsistencies** occurring across transformations that can cause errors. A simple example of this would a transformation that tries to access a column that has already been removed from a dataframe. A more complicated example is a transformation that assumes that a column has string values although the values have already been converted into integers. These inconsistencies are likely to appear due to the arbitrary order of execution of code cells allowed within notebooks.

As discussed in the introduction, we can also use the assumptions on the input data to address our other objectives like input data usage analysis, bias/skew detection, data provenance etc.

6 Implementation

The proposed dataframe graph abstract domain has been implemented as a part of PyLiSA[3] which is itself a Python front-end for the multi-language static analyser LiSA[4] based on abstract interpretation. LiSA is a completely modular static analysis framework that facilitates the easy implementation of a variety of abstract domains and support for several languages. Internally, LiSA translates the CFGs of the procedures of a program into a sequence of *symbolic expressions*. These symbolic expressions can be thought of as an *internal language* whose semantics are independent of the statements of the specific programming language that is being analysed. They can also help capture similar constructs across programming languages. For example, the same symbolic expression can be used to represent an array access although the syntax for array access can vary across languages. The analysis consists of an interprocedural analysis which runs the analysis over a procedure's CFG when it is called. The abstract state is composed of both memory and value abstraction. The memory and value abstract domains interpret the symbolic expressions and update their states according to the domain specific semantics of the expressions.

Within our implemention in PyLiSA, the symbolic expressions are a set of generic dataframe transformations similar to those presented in Table 1. PyLiSA is capable of parsing data science notebooks, extracting the code, and running our analysis to produce dataframe graphs corresponding to each dataframe variable. It also allows a user to specify the order of the execution of the cells within a notebooks and assembles the code accordingly. The next step within the implementation is to build the analysis over the graphs which extracts assumptions on the input data and also potentially warns the user of a possible erroneous transformation.

[3] `https://github.com/UniVE-SSV/pylisa`
[4] `https://github.com/UniVE-SSV/lisa`

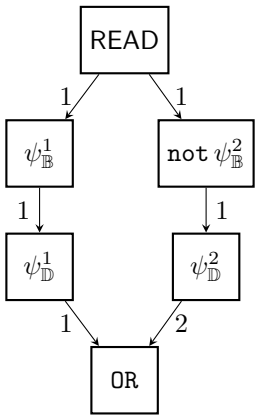


Figure 9: Graph produced after combining subgraphs of the graph in Figure 7

Graph optimizations While running the analysis, we often encounter that different subgraphs share nodes between each other, which allows us to potentially combine subgraphs when possible. The algorithms for the different lattice operations do not change with this optimization. It is also possible to further combine several sequential nodes into one in some cases. For example, when we have a straight chain of projection operations in the graph, it is possible to combine these into one single projection operation. Optimizations like these can help us drastically improve the visualization of the graphs.

Example 6.1. In the graph shown in Figure 7, we can combine the branches to produce a simpler graph as shown in Figure 9.

7 Conclusion

In this report, we have proposed a novel dataframe graph abstract domain that can track the transformations applied to data through a static analysis of data science notebooks. The produced dataframe graphs can be used to extract a variety of information about the shape of the dataframes and also collect the assumptions placed on the input data by the code. This information can be further used to detect transformations that can throw an error, detect unused data, and also check whether a transformation introduces bias or skew within the processed data. Since the data pre-processing stage is one of the first steps within the machine learning pipeline, it is crucial to ensure that there are no errors within the processed data since these can propagate into the trained models which would be much more difficult to debug. A final objective of this project would be an implementation of a tool integrated into the Jupyter notebook environment that can help a data scientist in real-time while writing code to transform data and signal potential problems.

References

[1] T. Herndon, M. Ash, and R. Pollin, "Does high public debt consistently stifle economic growth? A critique of Reinhart and Rogoff," *Cambridge Journal of Economics*, vol. 38, pp. 257–279, 12 2013.

[2] C. Urban and A. Miné, "A review of formal methods applied to machine learning," *CoRR*, vol. abs/2104.02466, 2021.

[3] I. Chen, F. D. Johansson, and D. Sontag, "Why is my classifier discriminatory?," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[4] P. Subotic, L. Milikic, and M. Stojic, "A static analysis framework for data science notebooks," *CoRR*, vol. abs/2110.08339, 2021.

[5] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, and Y. Wu, "Vamsa: Tracking provenance in data science scripts," *CoRR*, vol. abs/2001.01861, 2020.

[6] K. Yang, B. Huang, J. Stoyanovich, and S. Schelter, "Fairness-aware instrumentation of preprocessing pipelines for machine learning," *Workshop on Human-In-the-Loop Data Analytics (HILDA'20)*.

[7] C. Urban, "What programs want," *CoRR*, vol. abs/2007.10688, 2020.

[8] C. Urban and P. Müller, "An Abstract Interpretation Framework for Input Data Usage," in *ESOP*, pp. 683–710, 2018.

[9] S. Grafberger, S. Guha, J. Stoyanovich, and S. Schelter, *MLINSPECT: A Data Distribution Debugger for Machine Learning Pipelines*, p. 2736–2739. New York, NY, USA: Association for Computing Machinery, 2021.

[10] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, USA), p. 238–252, Association for Computing Machinery, 1977.

[11] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, (New York, NY, USA), p. 269–282, Association for Computing Machinery, 1979.

[12] P. Van Hentenryck, A. Cortesi, and B. Le Charlier, "Type analysis of prolog using type graphs," *SIGPLAN Not.*, vol. 29, p. 337–348, jun 1994.