Static Analysis for Data Scientists

Caterina Urban

▶ **To cite this version:**

Caterina Urban. Static Analysis for Data Scientists. Challenges of Software Verification, 238, Springer Nature Singapore, pp.77-91, 2023, Intelligent Systems Reference Library, 10.1007/978-981-19-9601-6_5 . hal-04249957

HAL Id: hal-04249957
https://inria.hal.science/hal-04249957

Submitted on 19 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Static Analysis for Data Scientists

Caterina Urban

Abstract Big data analytics has revolutionized the world of software development in the past decade. Every day, data scientists develop computer programs to gather, triage, and process data, in order to ultimately help us make data-driven decisions. As we rely more and more such data-manipulating software, we become increasingly vulnerable to poor choices, wrong assumptions, or other (programming or technical) mistakes made during software development. Mistakes that do not cause software failures can have serious consequences, since they give no indication that something went wrong along the way. In safety-critical applications, such mistakes can be deadly. In this chapter, we will present ongoing work to develop an abstract interpretation-based static analysis framework for data scientists. In particular, we will focus on issues arising from unexpected data and describe the challenges involved in designing and developing a practical static analysis that infers necessary expectations on the data read and manipulated using Jupyter notebooks, an increasingly popular development environment among data scientists.

1 Introduction

The advent of big data — the manipulation and analysis of massive quantities of data [15] – has revolutionized the world of software development in the past decade. Every day, *data scientists* [7] develop software programs to gather, triage, and pre-process data, which is varied, often unstructured, and generally "dirty" (i.e., inaccurate or even incorrect, incomplete, inconsistent, etc.). Data scientists have a mixed background in computer science and IT, and mathematics and statistics, as well as domain specific knowledge pertaining the type of data they work with (e.g., finance, biology, medicine, etc.). They are not professional software developers but, nonetheless, they spend most of their time writing software programs.

Caterina Urban
Inria & ENS | PSL, France, e-mail: caterina.urban@inria.fr

As we rely more and more on these data-manipulating programs for making decisions even in high stakes applications and sensitive applications (e.g., finance and medicine, but also hiring [16], credit scoring [10], prison sentencing [2], etc.), we become increasingly vulnerable to poor choices, wrong assumptions, or other mistakes (e.g., programming or technical errors) made during software development.

Mistakes that do not cause software failures can have serious consequences, since a plausible result gives no indication that something went wrong along the way. Just to cite a recent case in a medical application: a simple technical mistake made during data processing caused nearly 16 000 cases of Covid-19 between September 25th to October 2nd, 2020 to go unreported from official figures in the UK. As a consequence, Public Health England was unable to send out the relevant contact-tracing alerts [11]. Mistakes in medical applications can be deadly.

Jupyter notebooks are an increasingly popular development environment among data scientists [14]. They offer a read-eval-print loop (REPL) environment in which developers can quickly prototype code while interleaving textual descriptions and data visualizations (e.g., tables, charts, plots, etc.). Code cells in Jupyter notebooks can be (re-)executed in any desired order by the user, regardless of the order in which they are written. For this reason, the behavior of Jupyter notebooks is notoriously hard to predict and reproduce [20]. This makes prototyping and exploratory development *the most fragile phase of the data science development pipeline*. Uncaught fallacies at this phase can easily transfer to deployed code. It is also a recurrent phase of development, even after deployment, used for designing software customizations and updates to respond to discrete situational needs as new data becomes available.

The literature is scarce of work that aims at providing guarantees on the correctness of such data-manipulating software. A few static analysis approaches have been proposed to detect accidentally unused input data [19] and data leakages [18]. We focus here on issues arising from unexpected data, i.e., missing data, extra or duplicate data, data with a different format, etc.

1.1 Example

Let us consider the Jupyter notebook shown in Figure 1, which implements a simple course gradebook. Quiz grades are read from a CSV file in cell [2]. This yields a dataframe looking, for instance, as follows:

[2]:

	ID	Name	Q1	Q2	Q3
0	2394	Alice	A	A	A
1	4583	Bob	F	B	B
2	3956	Carol	F	A	C

In cell [3], letter grades (stored in the dataframe columns starting with the letter 'Q') are converted to a 4.0 GPA scale. Next, in cell [4], the average is computed and stored in a column 'Grade'. Student emails are read from a CSV in cell [5] and matched with the student grades in cell [6]. Finally, cell [7] retrieves student emails

```
[1]: import pandas as pd

[2]: df = pd.read_csv('Grades.csv')

[3]: grade2gpa = {'A': 4.0, 'B': 3.0, 'C': 2.0, 'D': 1.0, 'F': 0.0}
     quiz = df.columns.str.startswith('Q')
     df.iloc[:, quiz] = df.iloc[:, quiz].applymap(grade2gpa.get)

[4]: df['Grade'] = df.iloc[:, quiz].mean(axis=1)

[5]: es = pd.read_csv('Emails.csv')

[6]: un = pd.merge(df, es)

[7]: res = un[["Email", "Grade"]]
```

Fig. 1 Jupyter notebook implementing a simple course gradebook.

and their grade (to be used to send email notifications). In our example, this yields the following dataframe:

```
[7]:           Email  Grade
 0  alice@uni.eu    4.0
 1    bob@uni.eu    2.0
 2  carol@uni.eu    2.0
```

This notebook implicitly contains several expectations on the data it reads and manipulates. We show below two possible ways in which violating these expectations produces a wrong but plausible result.

Violation 1: Data with Missing Values.

Imagine the 'Grades.csv' file contains a missing value because a student was not present on the day of the quiz:

```
[2]:      ID   Name   Q1 Q2 Q3
 0  2394  Alice    A  A  A
 1  4583    Bob    F  B  B
 2  3956  Carol  NaN  A  C
```

The gradebook in Figure 1 does not take this case into account, i.e., *it expects that no quiz grades are missing*, and thus only computes the average over the quiz grades that are not missing:

```
[7]:           Email  Grade
 0  alice@uni.eu    4.0
 1    bob@uni.eu    2.0
 2  carol@uni.eu    3.0
```

Instead, it should probably replace missing values with 0.0 before computing the course grade in cell [4]:

[4]:
```
df.iloc[:, quiz] = df.iloc[:, quiz].fillna(0.0)
df['Grade'] = df.iloc[:, quiz].mean(axis=1)
```

Violation 2: Data with Different Format.

As another example, let us imagine that 'Grades.csv' also contains grades in a slightly different format:

[2]:

	ID	Name	Q1	Q2	Q3
0	2394	Alice	A	A	A
1	4583	Bob	F	B+	B
2	3956	Carol	F	A	C

The gradebook in Figure 1 does not take this case into account either, i.e., *it assumes that letter grades can only be 'A', 'B', 'C', 'D', or 'F'*. Thus, the 'B+' grade is treated as missing value:

[7]:

	Email	Grade
0	alice@uni.eu	4.0
1	bob@uni.eu	1.5
2	carol@uni.eu	2.0

Instead, a possible solution is to simply grades to remove any '+' or '-' symbol before converting letter grades:

[3]:
```
grade2gpa = {'A': 4.0, 'B': 3.0, 'C': 2.0, 'D': 1.0, 'F': 0.0}
quiz = df.columns.str.startswith('Q')
simplify = lambda x: x.strip('+-')
df.iloc[:, quiz] = df.iloc[:, quiz].applymap(simplify)
df.iloc[:, quiz] = df.iloc[:, quiz].applymap(grade2gpa.get)
```

In both of these cases the Jupyter notebook runs just fine, without raising any error. There is no indication that something went wrong along the way due to a mismatch between the data expectations implicit in the code and the actual data.

1.2 Data Expectation Static Analyses

The most widespread (and in many cases the only) method for ensuring software correctness is testing. However, thoroughly testing for data expectations is hard as it requires the test developer to be aware of them in the first place. Moreover, incentives for testing are low when Jupyter notebooks are disregarded as single-use non-critical code written as a means to an end.

In this chapter, we advocate for the need for *lightweight and practical static analyses* to automatically infer data expectations implicit in data-manipulation code in Jupyter notebooks. These static analyses should be directly *usable by data scientists*, without requiring any background in static analysis. Moreover, these analyses should be *interactive* to assist data scientists at all times while they develop and interact with their Jupyter notebooks.

We rely on the well-established framework of *abstract interpretation* [4] to:

(a) define a *concrete semantics* specifically tailored to indirectly reason about input data rather than only about program variables (cf. Section 2);
(b) design *abstract domains*, i.e., abstractions and algorithms to manipulate them, to correctly over-approximate the concrete semantics in a computable way (cf. Section 3);
(c) guide the practical *implementation* of these abstract semantics into usable static analyses for data scientists (cf. Section 4).

We will not present a fully-fledged solution but sketch ongoing work and focus the discussion on the challenges and opportunities that each of these steps brings along.

2 Input-Data Aware Concrete Semantics

2.1 Input Data

We consider *tabular data* stored, e.g., in CSV files. Let $\mathbb{S}$ be a set of string values. Furthermore, let $\mathbb{S}_{\mathsf{num}} \subseteq \mathbb{S}$ the sets of string values that can be interpreted as numerical values. We formalize a data file value or *dataframe* as a possibly empty $(r \times c)$-matrix of string values, where $r \in \mathbb{N}$ and $c \in \mathbb{N}$ denote the number of matrix rows and columns, respectively. We write ϵ to denote an empty dataframe. We assume that non-empty CSV files always have a header so the first row of non-empty dataframes contain the labels of the dataframe columns. Let

$$\mathbb{D} \stackrel{\text{def}}{=} \bigcup_{r \in \mathbb{N}} \bigcup_{c \in \mathbb{N}} \mathbb{S}^{r \times c} \tag{1}$$

be the set of all possible data file values. Let $D \in \mathbb{D} \setminus \{\epsilon\}$ be a non-empty dataframe. In the following, we write $\mathsf{hdr}(D)$ for the set of labels of the columns of D. Given a set of labels $C \subseteq \mathbb{S}$, we write $D[C]$ for the (sub)dataframe only containing the *columns* of D with labels in C. When C is a singleton $\{c\}$, with $c \in \mathbb{S}$, we simplify our notation and write $D[c]$ instead of $D[\{c\}]$. Given a dataframe with a single column V, we write $D[c] \bowtie V$, where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, for the (sub)dataframe only containing the *rows* of D that, in the column with label c, satisfy the value comparison with the corresponding rows of V. When all rows in V contain the same value v, we

$$
\begin{array}{lll}
\mathrm{A} ::= \mathrm{NaN} & & \\
\quad \mid s & & s \in \mathbb{S} \\
\quad \mid X[c] & & X \in \mathcal{X}, c \in \mathbb{S} \\
\quad \mid A_1 \diamond A_2 & & \diamond \in \{+, -, *, \div\} \\
\\
\mathrm{B} ::= A_1 \bowtie A_2 & & \bowtie \ \in \{<, \leq, =, \neq, >, \geq\} \\
\quad \mid B_1 \vee B_2 \mid B_1 \wedge B_2 & & \\
\\
\mathrm{E} ::= A & & \\
\quad \mid A_1 \ \textbf{if} \ B \ \textbf{else} \ A_2 & & \\
\\
\mathrm{S} ::= {}^{\ell}X := \textbf{input}() & & \ell \in \mathcal{L}, X \in \mathcal{X} \\
\quad \mid {}^{\ell}X_1 := X_2[C] & & \ell \in \mathcal{L}, X_1 \in \mathcal{X}, X_2 \in \mathcal{X}, C \subseteq \mathbb{S} \\
\quad \mid {}^{\ell}X_1 := X_2[c] \bowtie E & & \ell \in \mathcal{L}, X_1 \in \mathcal{X}, X_2 \in \mathcal{X}, c \in \mathbb{S} \\
\quad \mid {}^{\ell}X[c] := E & & \ell \in \mathcal{L}, X \in \mathcal{X}, c \in \mathbb{S} \\
\quad \mid {}^{\ell}X_1 := X_2 \multimap X_3 & & \ell \in \mathcal{L}, X_1 \in \mathcal{X}, X_2 \in \mathcal{X}, X_3 \in \mathcal{X} \\
\quad \mid S_1; S_2 & & \\
\\
\mathrm{P} ::= S^{\ell} & & \ell \in \mathcal{L}
\end{array}
$$

Fig. 2 Syntax of a toy language for data(frame) manipulation.

simplify our notation and write $D[c] \bowtie v$ instead of $D[c] \bowtie V$. We write $D|V$ for the dataframe resulting from adding column V to D. Finally, given two (non-empty) dataframes D_1 and D_2 with one ore more matching columns, we write $D_1 \multimap D_2$ for the dataframe resulting from (inner) joining D_1 and D_2.

2.2 Dataframe-Manipulating Language

We consider a toy programming language for data(frame) manipulation, which we use for illustration throughout the rest of the chapter. Let $\mathcal{X}$ be a finite set of dataframe variables. The syntax of program is defined inductively in Figure 2.

A program P consists of an instruction S followed by a unique label $\ell \in \mathcal{L}$. Another unique label appears before each instruction. In the following, given an instruction S, we write $\mathsf{lbl}(S)$ to denote the label of S.

The language contains instructions for reading an input CSV file ($X := \textbf{input}()$), and for selecting parts of a dataframe: the instruction $X_1 := X_2[C]$ only keeps dataframe columns with labels in a set $C \subseteq \mathbb{S}$, while the instruction $X_1 := X_2[c] \bowtie E$ only keeps dataframe rows with values (in the column with label c) that satisfy the condition. Conditions are value comparisons with NaN, a given string value $s \in \mathbb{S}$, or values in ($X[c]$) or resulting from operations on ($A_1 \diamond A_2$) other dataframe columns. They can also be conditional themselves (A_1 **if** B **else** A_2). Finally, the language contains instructions for replacing all values in a dataframe column ($X[c] := E$), for combining two dataframes by (inner) join ($X_1 := X_2 \multimap X_3$), and for concatenating two instructions together ($S_1; S_2$).

The language can be trivially extended to consider other ways to combine dataframes (e.g., concatenations, and left, right, or outer joins) as well as more complex instructions mimicking dataframe-manipulating operations of popular data science libraries. We also intentionally omitted the possibility of aliasing between dataframes, to keep things as simple as possible.

Gradebook Example

Here is the gradebook example in Figure 1 written in our toy language (simplified by assuming that the input CSV file only contains two quiz grades):

```
df := input()
df[Q1] := 4.0 if df[Q1]==A else
          (3.0 if df[Q1]==B else
          (2.0 if df[Q1]==C else
          (1.0 if df[Q1]==D else
          (0.0 if df[Q1]==F else NaN))))
df[Q2] := 4.0 if df[Q2]==A else
          (3.0 if df[Q2]==B else
          (2.0 if df[Q2]==C else
          (1.0 if df[Q2]==D else
          (0.0 if df[Q2]==F else NaN))))
df[Grade] := (df[Q1] + df[Q2]) ÷ 2
es := input()
un := df ⟜⟝ es
res := un[{Email, Grade}]

```

2.3 Input-Aware Semantics

We can now define the concrete semantics of data(frame)-manipulating programs.

Challenge

This semantics differs from the usual concrete semantics in that it must be **input data-aware**, that is, it must *perform a step of indirection to explicitly reason about data files* read by programs, in addition to reasoning about program variables.

An environment $\rho : \mathcal{X} \to \mathbb{D}$ maps each dataframe variable $X \in \mathcal{X}$ to its value $\rho(X) \in \mathbb{D}$. Let $\mathcal{E}$ denote the set of all environments. In addition, let $\delta : \mathcal{X} \mapsto \mathcal{P}(\mathcal{L})$ map dataframe variables to their data source, that is, the set of labels where dataframe value of the variable originates from. The data source of a dataframe variable can be a single label $\ell \in \mathcal{L}$, when the dataframe value of the variable originates from a dataframe read by the instruction with label ℓ, or a set of labels, when the dataframe

value originates from joining dataframes read at different instruction labels. Let Δ be the set of all possible data source maps. Finally, let $\phi\colon \mathcal{L} \to \mathcal{P}(\mathbb{D})$ map each instruction label to a possible data file value read at that label and let Φ be the set of all possible such maps. Environments keep track of dataframe variables, while data source and possible file value maps are what is needed to explicitly keep track of and reason about read data files.

The semantics of an expression E is a function $[\![E]\!] : \mathcal{E} \to \mathbb{D}$ mapping an environment to the dataframe (column) value of the expression in the given environment:

$$
\begin{aligned}
[\![\mathrm{NaN}]\!]\,\rho &\stackrel{\text{def}}{=} \mathrm{NaN}\\
[\![s]\!]\,\rho &\stackrel{\text{def}}{=} s\\
[\![X[c]]\!]\,\rho &\stackrel{\text{def}}{=} \rho(X)[c]\\
[\![A_1 \diamond A_2]\!]\,\rho &\stackrel{\text{def}}{=} [\![A_1]\!]\,\rho \diamond [\![A_2]\!]\,\rho
\end{aligned}
$$

$$
\begin{aligned}
[\![A_1 \bowtie A_2]\!]\,\rho &\stackrel{\text{def}}{=} [\![A_1]\!]\,\rho \bowtie [\![A_2]\!]\,\rho\\
[\![B_1 \vee B_2]\!]\,\rho &\stackrel{\text{def}}{=} [\![B_1]\!]\,\rho \vee [\![B_2]\!]\,\rho\\
[\![B_1 \wedge A_2]\!]\,\rho &\stackrel{\text{def}}{=} [\![B_1]\!]\,\rho \wedge [\![B_2]\!]\,\rho
\end{aligned}
$$

$$
[\![A_1 \textbf{ if } B \textbf{ else } A_2]\!]\,\rho \stackrel{\text{def}}{=} \begin{cases} [\![A_1]\!]\,\rho & [\![B]\!]\,\rho\\ [\![A_2]\!]\,\rho & \text{otherwise} \end{cases}
$$

A single value NaN or $s \in \mathbb{S}$ represents a dataframe column in which all rows contain that same value. All operations between dataframe columns (arithmetic, comparisons, boolean, etc.) are performed independently for each row.

The semantics of programs $\Pi\,[\![P]\!] : \mathcal{L} \mapsto \mathcal{P}\,(\mathcal{E} \times \Delta \times \Phi)$ maps each instruction label to the set of all triples of possible environments, possible data sources, and possible data file values read up to the point when the program execution is at that label. We define this semantics *forwards*, starting from the first instruction label where all environments in $\mathcal{E}$ are possible but no data files have yet been read:

$$
\Pi\,[\![P]\!] = \Pi\,[\![S^{\ell}]\!] \stackrel{\text{def}}{=} \Pi\,[\![S]\!] \left(\lambda p. \begin{cases} \mathcal{E} \times \{\dot{\emptyset}\} \times \{\dot{\emptyset}\} & p = \mathsf{lbl}(S)\\ \text{undefined} & \text{otherwise} \end{cases} \right)
$$

In Figure 3, we define the semantics $\Pi\,[\![S]\!] : (\mathcal{L} \mapsto \mathcal{P}\,(\mathcal{E} \times \Delta \times \Phi)) \to (\mathcal{L} \mapsto \mathcal{P}\,(\mathcal{E} \times \Delta \times \Phi))$ of each instruction pointwise within $\mathcal{P}\,(\mathcal{E} \times \Delta \times \Phi)$: each function $\mathcal{S}\,[\![S]\!] : \mathcal{P}\,(\mathcal{E} \times \Delta \times \Phi) \to \mathcal{P}\,(\mathcal{E} \times \Delta \times \Phi)$ takes as input a set W of triples of environments, data sources, and data file values and outputs triples of possible environments, possible data sources, and possible data file values read up to the point when the program has executed S. Note that, when the instruction reads a CSV file ($X :=$ **input**()), all data file values are possible after executing the instruction. Instead, instructions that select part of a dataframe (e.g., $X_1 := X_2[C]$)

$$\mathcal{S}\,[\![^{\ell} X := \textbf{input}()]\!]\,W \stackrel{\text{def}}{=} \{(\rho[X \mapsto D],\, \delta[X \mapsto \{\ell\}],\, \phi[\ell \mapsto D]) \mid (\rho, \delta, \phi) \in W, D \in \mathbb{D}\}$$

$$\mathcal{S}\,[\![^{\ell} X_1 := X_2[C]]\!]\,W \stackrel{\text{def}}{=} \left\{ (\rho', \delta, \phi) \;\middle|\; \begin{array}{c} (\rho, \delta, \phi) \in W, \\ C \subseteq \mathsf{hdr}(\rho(X_2)), \\ C \subseteq H \cup (\mathsf{hdr}(\rho(X_2)) \setminus H) \end{array} \right\}$$

$$H \stackrel{\text{def}}{=} \bigcup_{\ell \in \delta(X_2)} \mathsf{hdr}(\phi(\ell))$$

$$\rho' \stackrel{\text{def}}{=} \rho[X_1 \mapsto \rho(X_2)[C]]$$

$$\mathcal{S}\,[\![^{\ell} X_1 := X_2[c] \bowtie E]\!]\,W \stackrel{\text{def}}{=} \left\{ (\rho', \delta[X_1 \mapsto \delta(X_2)], \phi) \;\middle|\; \begin{array}{c} (\rho, \delta, \phi) \in \overleftarrow{[\![E]\!]}W, \\ c \in \mathsf{hdr}(\rho(X_2)), \\ c \in \bigcup_{\ell \in \delta(X_2)} \mathsf{hdr}(\phi(\ell)) \end{array} \right\}$$

$$\rho' \stackrel{\text{def}}{=} \rho[X_1 \mapsto \rho(X_2)[c] \bowtie [\![E]\!]\rho]$$

$$\mathcal{S}\,[\![^{\ell} X[c] := E]\!]\,W \stackrel{\text{def}}{=} W_1 \cup W_2$$

$$W_1 \stackrel{\text{def}}{=} \left\{ (\rho[X \mapsto \rho(X)[\rho(X)[c] \mapsto [\![E]\!]\rho]],\, \delta,\, \phi) \;\middle|\; \begin{array}{c} (\rho, \delta, \phi) \in \overleftarrow{[\![E]\!]}W, \\ c \in \mathsf{hdr}(\rho(X)) \end{array} \right\}$$

$$W_2 \stackrel{\text{def}}{=} \left\{ (\rho[X_1 \mapsto \rho(X_1)\,\Big|\, \begin{array}{c} c \\ [\![E]\!]\rho \end{array}],\, \delta,\, \phi) \;\middle|\; \begin{array}{c} (\rho, \delta, \phi) \in \overleftarrow{[\![E]\!]}W, \\ c \notin \mathsf{hdr}(\rho(X)) \end{array} \right\}$$

$$\mathcal{S}\,[\![^{\ell} X_1 := X_2 \multimap X_3]\!]\,W \stackrel{\text{def}}{=} \{(\rho', \delta', \phi) \mid (\rho, \delta, \phi) \in W\}$$

$$\rho' \stackrel{\text{def}}{=} \rho[X_1 \mapsto \rho(X_2) \multimap \rho(X_3)]$$

$$\delta' \stackrel{\text{def}}{=} \delta[X_1 \mapsto \delta(X_2) \cup \delta(X_3)]$$

$$\mathcal{S}\,[\![S_1; S_2]\!]\,W \stackrel{\text{def}}{=} \mathcal{S}\,[\![S_2]\!] \circ \mathcal{S}\,[\![S_1]\!]\,W$$

Fig. 3 Input-Aware Concrete Semantics of Instructions

impose *expectations* on dataframe values (e.g., $C \subseteq \mathsf{hdr}(\rho(X_2))$) and thus *restrict the set of possible data file values* after executing the instruction. The function $\overleftarrow{[\![E]\!]} : \mathcal{P}(\mathcal{E}, \Delta, \Phi) \to \mathcal{P}(\mathcal{E}, \Delta, \Phi)$ refines a set W based on expression E:

$$\overleftarrow{[\![\mathrm{NaN}]\!]}W = \overleftarrow{[\![s]\!]}W \stackrel{\text{def}}{=} W$$

$$\overleftarrow{[\![X[c]]\!]}W \stackrel{\text{def}}{=} \left\{ (\rho, \delta, \phi) \;\middle|\; \begin{array}{c} (\rho, \delta, \phi) \in W, \\ c \in \mathsf{hdr}(\rho(X)), \\ c \in H \cup (\mathsf{hdr}(\rho(X)) \setminus H) \end{array} \right\}$$

$$H \stackrel{\text{def}}{=} \bigcup_{\ell \in \delta(X)} \mathsf{hdr}(\phi(\ell))$$

$$\overleftarrow{[\![E_1 \odot E_2]\!]}W \stackrel{\text{def}}{=} \overleftarrow{[\![E_2]\!]} \circ \overleftarrow{[\![E_1]\!]}W \qquad \odot \in \{\diamond, \bowtie, \vee, \wedge\}$$

$$\overleftarrow{[\![A_1 \textbf{ if } B \textbf{ else } A_2]\!]}W \stackrel{\text{def}}{=} \overleftarrow{[\![A_1]\!]} \circ \overleftarrow{[\![B]\!]}W \cap \overleftarrow{[\![A_2]\!]} \circ \overleftarrow{[\![B]\!]}W$$

Thus, $\Pi\,[\![S^{\ell}]\!]\,(\ell)$ characterizes all possible (expected) data files read by program S^{ℓ}.

Gradebook Example (Continue)

The concrete semantics of our toy gradebook example is the following:

$$
\begin{aligned}
^{1} &\mapsto \mathcal{E} \times \{\dot{\emptyset}\} \times \{\dot{\emptyset}\} \\
^{2} &\mapsto \{(\rho[\texttt{df} \mapsto D], \delta[\texttt{df} \mapsto \{^{1}\}], \phi[^{1} \mapsto D]) \mid (\rho, \delta, \phi) \in {}^{1}, D \in \mathbb{D}\} \\
^{3} &\mapsto \left\{ (\rho[\texttt{df} \mapsto \rho(\texttt{df})[\rho(\texttt{df})[Q1] \mapsto [\![\texttt{4.0} \ldots \texttt{NaN}]\!]\rho]], \delta, \phi) \;\middle|\; \begin{array}{c} (\rho, \delta, \phi) \in {}^{2} \\ \text{Q1} \in \mathsf{hdr}(\rho(\texttt{df})) \\ \text{Q1} \in \mathsf{hdr}(\phi(^{1})) \end{array} \right\} \\
^{4} &\mapsto \left\{ (\rho[\texttt{df} \mapsto \rho(\texttt{df})[\rho(\texttt{df})[Q2] \mapsto [\![\texttt{4.0} \ldots \texttt{NaN}]\!]\rho]], \delta, \phi) \;\middle|\; \begin{array}{c} (\rho, \delta, \phi) \in {}^{3} \\ \text{Q2} \in \mathsf{hdr}(\rho(\texttt{df})) \\ \text{Q2} \in \mathsf{hdr}(\phi(^{1})) \end{array} \right\} \\
^{5} &\mapsto \left\{ \left(\rho[\texttt{df} \mapsto \rho(\texttt{df}) \;\middle|\; \begin{array}{c} \texttt{Grade} \\ [\![(\texttt{df[Q1]} + \texttt{df[Q2]}) \div 2]\!]\rho \end{array}], \delta, \phi \right) \;\middle|\; (\rho, \delta, \phi) \in {}^{4} \right\} \\
^{6} &\mapsto \{(\rho[\texttt{es} \mapsto D], \delta[\texttt{es} \mapsto \{^{5}\}], \phi[^{5} \mapsto D]) \mid (\rho, \delta, \phi) \in {}^{5}, D \in \mathbb{D}\} \\
^{7} &\mapsto \{(\rho[\texttt{un} \mapsto \rho(\texttt{df}) \multimap \rho(\texttt{es})], \delta[\texttt{un} \mapsto \{^{1}, {}^{5}\}], \phi) \mid (\rho, \delta, \phi) \in {}^{6}\} \\
^{8} &\mapsto \left\{ (\rho[\texttt{un} \mapsto \rho(\texttt{un})[\{\texttt{Email}, \texttt{Grade}\}]], \delta, \phi) \;\middle|\; \begin{array}{c} (\rho, \delta, \phi) \in {}^{7} \\ \{\texttt{Email}, \texttt{Grade}\} \subseteq \mathsf{hdr}(\rho(\texttt{un})) \\ \texttt{Email} \in \mathsf{hdr}(\phi(^{1})) \cup \mathsf{hdr}(\phi(^{5})) \end{array} \right\}
\end{aligned}
$$

Note that, for simplicity, we only considered the case in which the column 'Grade' was not already present in the data file read at instruction label ^{1}.

---

3 Expectations Abstract Domains

We now design a decidable abstraction of $\Pi\,[\![P]\!]$ which *over-approximates* the concrete semantics of P at each instruction label $\ell \in \mathcal{L}$. As a consequence, this abstraction yields *necessary* expectations on data frame values for a program to execute successfully and correctly. In particular, if a data file value is not in the abstraction, the program will definitely eventually run into an error or compute a wrong result if it tries to read data from it. On the other hand, if a data file value is in the abstraction there is no guarantee that the program will execute successfully and correctly when reading data from it. This choice is intentional so as to provide immediately actionable results to data scientists rather than overwhelm them with possible false negatives: a mismatch between a data file value and (the abstraction of) a program indicates something that must be corrected (either in the program or the data file).

The abstraction $\Pi^{\natural}\,[\![P]\!] : \mathcal{L} \to \mathcal{W}$ associates to each instruction label $\ell \in \mathcal{L}$ and element $W^{\natural} \in \mathcal{W}$ of an abstract domain $\mathbb{W}$. $W^{\natural}$ over-approximates the possible environments, possible data sources, and possible data file values read up to the point when the program execution has reached the instruction with label ℓ.

Challenge

The main challenge in designing such an abstraction is that it must reason about **multi-dimensional data structures** such as dataframes, rather than simpler values.

3.1 Column Expectations Abstract Domain

As a simple example, we sketch an abstraction that infers expectations about column labels that a data file value *must* have. The elements of the column expectations abstract domain $\mathbb{C}$ belong to a set C of triples. The first element of each of these triples is an abstract environment $\rho^\natural \colon \mathcal{X} \to \mathcal{P}(\mathbb{S}) \times \mathcal{P}(\mathbb{S})$ mapping variables to sets of column labels that its dataframe value *may not* have and *must* have, respectively. Notably, we will use the "may not" column labels set to track columns that were potentially added by a program and thus were not necessarily present in the original data source. Given an abstract environment $\rho^\natural \in \mathcal{E}^\natural$ and a variable $X \in \mathcal{X}$, we write $\rho^\natural_m(X)$ for the "may not" set associated with X in $\rho^\natural$, and $\rho^\natural_M(X)$ for the "must" set. The second element of these triples is an abstract data source map $\delta^\natural \colon \mathcal{X} \to \mathcal{P}(\mathcal{L})$ mapping dataframe variables to their potential data sources. Finally, the third element is an abstract data file value map $\phi^\natural \colon \mathcal{L} \to \mathcal{P}(\mathbb{S})$ mapping each instruction label to the set of columns that the data file read at that label *must* have.

The concretization function $\gamma_{\mathbb{C}} \colon C \to \mathcal{P}(\mathcal{E} \times \Delta \times \Phi)$ is defined as follows:

$$\gamma_{\mathbb{C}}((\rho^\natural, \delta^\natural, \phi^\natural)) \stackrel{\text{def}}{=} \left\{ (\rho, \delta, \phi) \in \mathcal{E} \times \Delta \times \Phi \;\middle|\; \begin{array}{c} \forall X \in \mathcal{X} \colon \rho^\natural_M(X) \subseteq \mathsf{hdr}(\rho(X)) \\ \forall X \in \mathcal{X} \colon \delta(X) = \delta^\natural(X), \\ \forall \ell \in \mathcal{L} \colon \phi^\natural(\ell) \subseteq \mathsf{hdr}(\phi(\ell)) \end{array} \right\}$$

To define the abstract semantics of programs $\Pi \, [\![P]\!]^\natural : \mathcal{L} \mapsto C$, we first define the function $\overleftarrow{[\![E]\!]}^\natural \colon C \to C$ refining an abstract element $C^\natural \in C$ based on the expression E (and abstracting the concrete refinement function $\overleftarrow{[\![E]\!]}$):

$$\overleftarrow{[\![\mathrm{NaN}]\!]}^\natural C^\natural = \overleftarrow{[\![s]\!]}^\natural \stackrel{\text{def}}{=} C^\natural$$

$$\overleftarrow{[\![X[c]]\!]}^\natural(\rho^\natural, \delta^\natural, \phi^\natural) \stackrel{\text{def}}{=} (\rho^\natural[X \mapsto (\rho^\natural_m(X), \rho^\natural_M(X) \cup \{c\})], \delta^\natural, \phi^{\natural\prime})$$

$$\phi^{\natural\prime} \stackrel{\text{def}}{=} \begin{cases} \phi^\natural[\ell' \mapsto \phi^\natural(\ell') \cup (\{c\} \setminus \rho^\natural_m(X))] & \delta^\natural(X) = \{\ell'\} \\ \phi^\natural & \text{otherwise} \end{cases}$$

$$\overleftarrow{[\![E_1 \odot E_2]\!]}^\natural C^\natural \stackrel{\text{def}}{=} \overleftarrow{[\![E_2]\!]}^\natural \circ \overleftarrow{[\![E_1]\!]}^\natural C^\natural \qquad \odot \in \{\diamond, \bowtie, \vee, \wedge\}$$

$$\overleftarrow{[\![A_1 \textbf{ if } B \textbf{ else } A_2]\!]}^\natural C^\natural \stackrel{\text{def}}{=} \overleftarrow{[\![A_1]\!]}^\natural \circ \overleftarrow{[\![B]\!]}^\natural C^\natural \oplus \overleftarrow{[\![A_2]\!]}^\natural \circ \overleftarrow{[\![B]\!]}^\natural C^\natural$$

A column selection expression $X[c]$ adds the column c to the "must" set of column labels for X in the abstract environment; column c is also added to the data source values of X, if this can be traced back to a single data source, and column c is not potentially added by the program. If X cannot be traced back to a single data source, there is a potential loss of precision in tracking column labels. In case of a

$$\mathcal{S}^\natural[\![^\ell X := \textbf{input}()]\!](\rho^\natural, \delta^\natural, \phi^\natural) \stackrel{\text{def}}{=} (\rho^\natural[X \mapsto (\emptyset, \emptyset)], \delta^\natural[X \mapsto \ell], \phi^\natural[\ell \mapsto \emptyset])$$

$$\mathcal{S}^\natural[\![^\ell X_1 := X_2[C]]\!](\rho^\natural, \delta^\natural, \phi^\natural) \stackrel{\text{def}}{=} (\rho^{\natural\prime}, \delta^\natural[X_1 \mapsto \delta^\natural(X_2)], \phi^{\natural\prime})$$

$$\rho^{\natural\prime} \stackrel{\text{def}}{=} \rho^\natural[X_1 \mapsto (\rho^\natural_m(X_2) \cap C, C)]$$

$$\phi^{\natural\prime} \stackrel{\text{def}}{=} \begin{cases} \phi^\natural[\ell' \mapsto \phi^\natural(\ell') \cup (C \setminus \rho^\natural_m(X_2))] & \delta^\natural(X_2) = \{\ell'\} \\ \phi^\natural & \text{otherwise} \end{cases}$$

$$\mathcal{S}^\natural[\![^\ell X_1 := X_2[c] \bowtie E]\!] C^\natural \stackrel{\text{def}}{=} (\rho^{\natural\prime}, \delta^\natural[X_1 \mapsto \delta^\natural(X_2)], \phi^{\natural\prime}) \text{ where } (\rho^\natural, \delta^\natural, \phi^\natural) = \overleftarrow{[\![E]\!]} C^\natural$$

$$\rho^{\natural\prime} \stackrel{\text{def}}{=} \rho^\natural[X_1 \mapsto (\rho^\natural_m(X_2), \rho^\natural_{\boldsymbol{M}}(X_2) \cup \{c\})]$$

$$\phi^{\natural\prime} \stackrel{\text{def}}{=} \begin{cases} \phi^\natural[\ell' \mapsto \phi^\natural(\ell') \cup (\{c\} \setminus \rho^\natural_m(X_2))] & \delta^\natural(X_2) = \{\ell'\} \\ \phi^\natural & \text{otherwise} \end{cases}$$

$$\mathcal{S}^\natural[\![^\ell X[c] := E]\!] C^\natural \stackrel{\text{def}}{=} (\rho^{\natural\prime}, \delta^\natural, \phi^\natural) \text{ where } (\rho^\natural, \delta^\natural, \phi^\natural) = \overleftarrow{[\![E]\!]} C^\natural$$

$$\rho^{\natural\prime} \stackrel{\text{def}}{=} \rho^\natural[X \mapsto (\rho^\natural_m(X) \cup (\{c\} \setminus \rho^\natural_{\boldsymbol{M}}(X)), \rho^\natural_{\boldsymbol{M}}(X) \cup \{c\})]$$

$$\mathcal{S}^\natural[\![^\ell X_1 := X_2 \multimap X_3]\!](\rho^\natural, \delta^\natural, \phi^\natural) \stackrel{\text{def}}{=} (\rho^{\natural\prime}, \delta^\natural[X_1 \mapsto \delta^\natural(X_2) \cup \delta^\natural(X_3)], \phi^\natural)$$

$$\rho^{\natural\prime} \stackrel{\text{def}}{=} \rho^\natural[X_1 \mapsto (\rho^\natural_m(X_2) \cup \rho^\natural_m(X_3), \rho^\natural_{\boldsymbol{M}}(X_2) \cup \rho^\natural_{\boldsymbol{M}}(X_3))]$$

$$\mathcal{S}^\natural[\![S_1; S_2]\!] C^\natural \stackrel{\text{def}}{=} \mathcal{S}^\natural[\![S_2]\!] \circ \mathcal{S}^\natural[\![S_1]\!] C^\natural$$

Fig. 4 Column Expectations Abstract Semantics of Instructions

conditional expressions, the abstract triples refined by the two conditional branches are merged together taking the intersection of corresponding sets of labels. (Note that "may not" sets in abstract environments and abstract data source maps are never modified by this refining function. The only refinements happen in "must" set of labels in abstract environments and abstract data source value maps.)

Thus, the abstract semantics of programs $\Pi[\![P]\!]^\natural : \mathcal{L} \mapsto \mathcal{E}^\natural \times \Delta^\natural \times \Phi^\natural$ maps each instruction label to an abstract domain element:

$$\Pi[\![P]\!]^\natural = \Pi[\![S^\ell]\!]^\natural \stackrel{\text{def}}{=} \Pi[\![S]\!]^\natural \left(\lambda p. \begin{cases} (\lambda X \in \mathcal{X} : (\emptyset, \emptyset), \dot{\emptyset}, \dot{\emptyset}) & p = \mathsf{lbl}(S) \\ \bot_{\mathbb{C}} & \text{otherwise} \end{cases} \right)$$

where the abstract semantics $\mathcal{S}[\![S]\!]^\natural : (\mathcal{L} \mapsto C) \to (\mathcal{L} \mapsto C)$ of each instruction S is defined pointwise within C in Figure 4.

Gradebook Example (Continue)

The abstract semantics of our toy gradebook example is the following:

$^1 \mapsto (\lambda X \in \mathcal{X} : (\emptyset, \emptyset), \dot{\emptyset}, \dot{\emptyset})$

$^2 \mapsto (\rho^\natural[\texttt{df} \mapsto (\emptyset, \emptyset)], \delta^\natural[\texttt{df} \mapsto \{^1\}], \phi^\natural[^1 \mapsto \emptyset])$ where $(\rho^\natural, \delta^\natural, \phi^\natural) = {}^1$

$^3 \mapsto (\rho^\natural[\texttt{df} \mapsto (\emptyset, \{\text{Q1}\})], \delta^\natural, \phi^\natural[^1 \mapsto \{\text{Q1}\}])$ where $(\rho^\natural, \delta^\natural, \phi^\natural) = {}^2$

$^4 \mapsto (\rho^\natural[\texttt{df} \mapsto (\emptyset, \{\text{Q1}, \text{Q2}\})], \delta^\natural, \phi^\natural[^1 \mapsto \{\text{Q1}, \text{Q2}\}])$ where $(\rho^\natural, \delta^\natural, \phi^\natural) = {}^3$

$^5 \mapsto (\rho^\natural[\texttt{df} \mapsto (\{\text{Grade}\}, \{\text{Q1}, \text{Q2}, \text{Grade}\})], \delta^\natural, \phi^\natural)$ where $(\rho^\natural, \delta^\natural, \phi^\natural) = {}^4$

^{6} $\mapsto$ $(\rho^\natural[\texttt{es} \mapsto (\emptyset, \emptyset)], \delta^\natural[\texttt{es} \mapsto \{^{5}\}], \phi^\natural[^{5} \mapsto \emptyset])$ where $(\rho^\natural, \delta^\natural, \phi^\natural) = {}^{5}$
^{7} $\mapsto$ $(\rho^\natural[\texttt{un} \mapsto (\{\text{Grade}\}, \{\text{Q1}, \text{Q2}, \text{Grade}\})], \delta^\natural[\texttt{un} \mapsto \{^{1}, ^{5}\}], \phi^\natural)$ where $(\rho^\natural, \delta^\natural, \phi^\natural) = {}^{6}$
^{8} $\mapsto$
$(\rho^\natural[\texttt{res} \mapsto (\{\text{Grade}\},\{\text{Email}, \text{Grade}\})], \delta^\natural[\texttt{res} \mapsto \{^{1}, ^{5}\}], \phi^\natural)$ where $(\rho^\natural, \delta^\natural, \phi^\natural) = {}^{7}$

Note the loss of precision in tracking column labels after the dataframe join.

3.2 Other Expectations Abstract Domains

Several other such abstract domains can be defined to track expectations about, e.g., data types or data values. In our toy gradebook examples, we could infer that values in columns "Q1" and "Q2" are expected to be strings in { 'A', 'B', 'C', 'D', 'F' }. Existing numerical domains [6, 12, 13, etc.], string domains [3, 1, etc.], and abstract domains for data structures [5, 9, etc.] can be more or less easily adapted to work in this settings. By building upon relational abstract domains, one can even track relationships between data columns or values. Of course, the more sophisticated data expectations one wants to infer, the more complex the abstract domain definition will be.

4 Implementation

Our implementation of these data expectation static analyses is ongoing, targeting Jupyter notebooks. We want these to be practically useful and directly usable by data scientists, without requiring them to have any background in static analysis.

For the moment, we are developing our static analyses for Jupyter notebooks using Python. In the long term, we want to support other languages used for data science such as R, as well as the not uncommon practice of using multiple programming languages in the same notebooks.

! Challenge

The challenge with such a long term goal is, not only developing static analyses for *dynamic languages* such as Python or R (with their complex data science libraries), but also *their combination*, taking into account their underlying practical semantic differences (e.g., array indexing starting at 0 in Python but at 1 in R, missing values automatically ignored in Python but not in R, etc.).

We are also studying combinations of dynamic and static analyses to effectively deal with really dynamic features of these languages (e.g., `eval()` expressions in Python). In particular, we are looking into using dynamic executions to guide the static analysis in a principled way, akin to abstract conflict driven learning [8].

Finally, to maximize usability, we aim to integrate our static analyses into the Jupyter notebook environment, either as extensions or integrating them in the most used integrated development environment for Jupyter notebooks such as Visual Studio Code, PyCharm, etc.

Challenge

The main challenge to develop really useful static analyses for data scientist is to render them *interactive* [17] to adapt them to the way data scientist write and use their Jupyter notebooks. This also means that the analyses should be sufficiently *lightweight* to be able to compute results quickly, but at the same time precise enough for the results to remain useful in practice.

5 Conclusion

In this chapter, we have argued for the need to develop new static analyses tailored for Jupyter notebooks and directly usable by data scientists without a static analysis background. We have sketched a simple static analysis framework to infer expectations about the data manipulated by a Jupyter notebook and highlighted the challenges that come with making such static analyses a reality in the near future.

More generally, ours is a long-term effort to democratize static analysis, to apply it to a wider range of software, and render it more accessible to a broader audience. We hope that others will join us in this endeavour.

Acknowledgements. Work partly supported by DAIS, Ca' Foscari University of Venice, within the 383 IRIDE project 'Static Analysis of Notebook Python' directed by A. Cortesi.

References

1. V. Arceri, M. Olliaro, A. Cortesi, and P. Ferrara. Relational String Abstract Domains. In *VMCAI*, pages 20–42, 2022.
2. A. Chouldechova. Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments. *Big Data*, 5(2):153–163, 2017.
3. G. Costantini, P. Ferrara, and A. Cortesi. A suite of abstract domains for static analysis of string values. *Software - Practice and Experience*, 45(2):245–287, 2015.
4. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
5. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.
6. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.

7. T. H. Davenport and D. J. Patil. Data Scientist: The Sexiest Job of the 21st Century. *Harvard Business Review*, 90(10):70–76, October 2012.
8. V. V. D'Silva, L. Haller, and D. Kroening. Abstract Conflict Driven Learning. In *POPL*, pages 143–154, 2013.
9. J. Fulara. Generic Abstraction of Dictionaries and Arrays. *Electronic Notes in Theoretical Computer Science*, 287:53–64, 2012.
10. A. E. Khandani, A. J. Kim, and A. W. Lo. Consumer Credit-Risk Models via Machine-Learning Algorithms. *Journal of Banking & Finance*, 34(11):2767–2787, 2010.
11. E. Mahase. Covid-19: Only Half of 16 000 Patients Missed from England's Official Figures Have Been Contacted. *BMJ*, 371, 2020.
12. A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, 2004.
13. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
14. J. M. Perkel. Why Jupyter is Data Scientists' Computational Notebook of Choice. *Nature*, 563(7729):145–146, November 2018.
15. S. Sagiroglu and D. Sinanc. Big Data: A Review. In *CTS*, pages 42–47, 2013.
16. C. Schumann, J. S. Foster, N. Mattei, and J. P. Dickerson. We Need Fairness and Explainability in Algorithmic Hiring. In *AAMAS*, pages 1716–1720, 2020.
17. B. Stein, B. E. Chang, and M. Sridharan. Demanded abstract interpretation. In *PLDI*, pages 282–295, 2021.
18. P. Subotic, U. Bojanic, and M. Stojic. Statically Detecting Data Leakages in Data Science Code. In *SOAP*, pages 16–22, 2022.
19. C. Urban and P. Müller. An Abstract Interpretation Framework for Input Data Usage. In A. Ahmed, editor, *ESOP*, pages 683–710, 2018.
20. J. Wang, L. Li, and A. Zeller. Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks. In G. Rothermel and D. Bae, editors, *ICSE-NIER*, pages 53–56, 2020.