What Programs Want
Automatic Inference of Input Data Specifications

Caterina Urban[1,2]

[1] INRIA, Paris, France
[2] DIENS, École Normale Supérieure, CNRS, PSL University, Paris, France
caterina.urban@inria.fr

Abstract. Nowadays, as machine-learned software quickly permeates our society, we are becoming increasingly vulnerable to programming errors in the data pre-processing or training software, as well as errors in the data itself. In this paper, we propose a static shape analysis framework for input data of data-processing programs. Our analysis automatically infers necessary conditions on the structure and values of the data read by a data-processing program. Our framework builds on a family of underlying abstract domains, extended to indirectly reason about the input data rather than simply reasoning about the program variables. The choice of these abstract domain is a parameter of the analysis. We describe various instances built from existing abstract domains. The proposed approach is implemented in an open-source static analyzer for PYTHON programs. We demonstrate its potential on a number of representative examples.

1 Introduction

Due to the availability of vast amounts of data and corresponding tremendous advances in machine learning, computer software is nowadays an ever increasing presence in every aspect our society. As we rely more and more on machine-learned software, we become increasingly vulnerable to programming errors but (in contrast to traditional software) also errors in the data used for training.

In general, before software training, the data goes through long pre-processing pipelines[3]. Errors can be missed, or even introduced, at any stage of these pipelines. This is even more true when data pre-processing stages are disregarded as single-use glue code and, for this reason, are poorly tested, let alone statically analyzed or verified. Moreover, this kind of code is often written in a rush and is highly dependent on the data (e.g., the use of magic constants is not uncommon) All this together, greatly increases the likelihood for errors to be noticed extremely late in the pipeline (which entails a more or less important waste of time), or more dangerously, to remain completely unnoticed.

[3] https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html

```
1  grade2gpa = { 'A': 4.0, 'B': 3.0, 'C': 2.0, 'D': 1.0, 'F': 0.0 }
2  students = int(input())
3  for _ in range(students):
4      name = input()
5      classes = int(input())
6      gpa = 0.0
7      for _ in range(classes):
8          grade = input()
9          gpa += grade2gpa[grade]
10     result = gpa / classes
11     print('{}: {}'.format(name, result))
```

Fig. 1: Simple GPA calculator for multiple students.

Motivating Example. As an example, let us consider the data processing PYTHON code shown in Figure 1, which calculates the simple GPA for a given number of students (cf. Line 2). For each class taken by a student (cf. Line 7), their (A-F) grade is converted into a numeric (4-0) grade, and all numeric grades are added together (cf. Line 9). The GPA is obtained by dividing this by the number of classes taken by the student (cf. Line 10).

Even this small program makes several assumptions on its input data. For instance, it assumes that the very first input read by the program (cf. Line 2) is a string representation of an integer number that indicates how many student records follow in the data file (cf. Line 3). A similar assumption holds for the second input read for each student record (cf. Line 5), which should indicate how many student grades follow in the data file (cf. Line 7). This number should be different from zero (or the division at Line 10 would raise a ZeroDivisionError). Finally, the program assumes that each grade read at Line 8 is a string in the set {'A', 'B', 'C', 'D', 'F'} (or the dictionary access at Line 9 would raise a KeyError). Note that, not all assumptions necessarily lead to a program error if violated. For instance, consider the following data stream:

1 Emma 1 A F
↑

A mistake is indicated by the arrow: the number of classes taken by the student Emma is off by one (i.e., it should be 2 instead of 1). In this case the program in Figure 1 will not raise any error but will instead compute a wrong (but plausible!) GPA for Emma (i.e., 4.0 instead of 2.0).

Our Approach. To address these issues, we propose an abstract interpretation-based *shape analysis* framework *for input data* of data-processing programs. The analysis automatically infers implicit assumptions on the input data that are embedded in the source code of a program. Specifically, we infer assumptions on the structure of the data as well as on the values and the relations between the data.

We propose a *new* data shape *abstract domain*, capable of reasoning about the input data in addition to the program variables. The domain builds on a family of underlying over-approximating abstract domains, which collect constraints

on the program variables and, indirectly, on the input data of a program. The abstract domain is parametric in the choice of the underlying domains.

Thus, our analysis infers *necessary conditions* on the data read by the program, i.e., conditions such that, if violated, guarantee that the program will execute unsuccessfully or incorrectly. This approach suffers from false negatives. However, we argue that this is preferable in practice to overwhelming data scientists with possibly many false positives (as with sufficient conditions).

Back to our motivating example, the analysis (parameterized by the sign abstract domain [13] and the finite string set domain [9]) infers that data files read by the program in Figure 1 have the following shape:

$$d_1 \begin{cases} 1 & \text{INT} \geq 0 \\ 2 & \text{STRING} \\ 3 & \text{INT} \geq 0 \\ d_3 \begin{cases} 4 & \text{STRING} \in \{\text{'A', 'B', 'C', 'D', 'F'}\} \\ \vdots & \dots \end{cases} \\ \vdots & \end{cases}$$

where d_i denotes the data at line i of the data file. Thus, the analysis would detect the mistake discussed above, since a data file containing the erroneous data does not match this inferred condition.

Note that, in general, a mismatch between a data file and a data-processing program indicates a mistake either in data or in the source code of the program. Our analysis does not aim to address this question. More generally, the result of our analysis can be used for a wide range of applications: from code specifications [14], to grammar-based testing [29], to automatically checking and guiding the cleaning of the data [1,38].

Outline. Section 2 introduces the syntax and concrete semantics of our data-processing programs. In Section 3, we define and present instances of the underlying abstract domains. We describe the rest our data shape abstract domain in Section 4 and define the abstract semantics in Section 5. Our prototype static analyzer is presented in 6. Finally, Section 7 discusses related work and Section 8 concludes and envisions future work.

2 Input Data-Aware Program Semantics

Input Data. We consider *tabular data* stored, e.g., in CSV files. We note, however, that what we present easily generalizes to other files as, e.g., spreadsheets.

Let $\mathbb{S}$ be a set of string values. Furthermore, let $\mathbb{S}_{\mathsf{int}} \subseteq \mathbb{S}$ and $\mathbb{S}_{\mathsf{float}} \subseteq \mathbb{S}$ be the sets of string values that can be interpreted as integer and float values, respectively. We formalize a data file 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 row (i.e., data

$$
\begin{aligned}
\text{A} ::=\ & X && X \in \mathcal{X} \\
\mid\ & v && v \in \mathbb{V} \\
\mid\ & \textbf{input}() \mid \textbf{int}(A) \mid \textbf{float}(A) \\
\mid\ & A_1 \diamond A_2 && \diamond \in \{+, -, *, /\} \\
\text{B} ::=\ & A_1 \bowtie A_2 && \bowtie \in \{<, \leq, =, \neq, >, \geq\} \\
\mid\ & B_1 \vee B_2 \mid B_1 \wedge B_2 \\
\text{S} ::=\ & {}^{l}X := A && l \in \mathcal{L}, X \in \mathcal{X} \\
\mid\ & \textbf{if } {}^{l}B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} && l \in \mathcal{L} \\
\mid\ & \textbf{for } {}^{l}A \textbf{ do } S \textbf{ od} \mid \textbf{while } {}^{l}B \textbf{ do } S \textbf{ od} && l \in \mathcal{L} \\
\mid\ & S_1; S_2 \\
\text{P} ::=\ & S^{l} && l \in \mathcal{L}
\end{aligned}
$$

Fig. 2: Syntax

records) and columns (i.e., data fields), respectively. We write ϵ to denote an empty data file. Let

$$\mathcal{D} \stackrel{\text{def}}{=} \bigcup_{r \in \mathbb{N}} \bigcup_{c \in \mathbb{N}} \mathbb{S}^{r \times c} \tag{2.1}$$

be the set of all data files. Without loss of generality, to simplify our formalization, we assume that data records contain only one field, i.e., $r = 1$. We lift this assumption and consider multiple data fields in Section 3.2.

Data-Processing Language. We consider a toy PYTHON-like programming language for data manipulation, which we use for illustration throughout the rest of the paper. Let $\mathcal{X}$ be a finite set of program variables, and let $\mathbb{V} \stackrel{\text{def}}{=} \mathbb{Z} \cup \mathbb{F} \cup \mathbb{S}$ be a set of values partitioned in sets of integer ($\mathbb{Z}$), float ($\mathbb{F}$), and string ($\mathbb{S}$) values. The syntax of programs is defined inductively in Figure 2. A program P consists of an instruction S followed by a unique label $l \in \mathcal{L}$. Another unique label appears within each instruction. Programs can read data from an input data file: the **input**() expression consumes a record from the input data file. Without loss of generality, to simplify our formalization, we assume that only the right-hand sides of assignments can contain **input**() sub-expressions. (Programs can always be rewritten to satisfy this assumption.) The **for** A **do** S **od** instruction repeats an instruction S for A times. The rest of the language syntax is standard.

Input-Aware Semantics. We can now define the (concrete) semantics of the data-processing programs. This semantics differs from the usual semantics in that it is *input data-aware*, that is, it explicitly considers the data read by programs.

An environment $\rho\colon \mathcal{X} \to \mathbb{V}$ maps each program variable $X \in \mathcal{X}$ to its value $\rho(X) \in \mathbb{V}$. Let $\mathcal{E}$ denote the set of all environments.

The semantics of an arithmetic expression A is a function $\mathcal{A}[\![A]\!] : \mathcal{E} \times \mathcal{D} \to \mathbb{V} \times \mathcal{D}$ mapping an environment and a data file to the value (in $\mathbb{V}$) of the expression

$$\mathcal{S}\left[\!\left[{}^{l}X := A\right]\!\right] W \stackrel{\text{def}}{=} \{\langle \rho, D\rangle \in \mathcal{E} \times \mathcal{D} \mid \langle v, R\rangle = \mathcal{A}[\![A]\!]\langle \rho, D\rangle, \langle \rho[X \mapsto v], R\rangle \in W\}$$

$$\mathcal{S}\left[\!\left[\textbf{if } {}^{l}B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\right]\!\right] W \stackrel{\text{def}}{=} W1 \cup W2$$

$$W1 \stackrel{\text{def}}{=} \{\langle \rho, D\rangle \in \mathcal{E} \times \mathcal{D} \mid \mathsf{tt} = \mathcal{B}[\![B]\!]\rho, \langle \rho, D\rangle \in \mathcal{S}[\![S_1]\!] W\}$$

$$W2 \stackrel{\text{def}}{=} \{\langle \rho, D\rangle \in \mathcal{E} \times \mathcal{D} \mid \mathsf{ff} = \mathcal{B}[\![B]\!]\rho, \langle \rho, D\rangle \in \mathcal{S}[\![S_2]\!] W\}$$

$$\mathcal{S}\left[\!\left[\textbf{for } {}^{l}A \textbf{ do } S \textbf{ od}\right]\!\right] W \stackrel{\text{def}}{=} \{\langle \rho, D\rangle \in \mathcal{E} \times \mathcal{D} \mid \langle 0, D\rangle = \mathcal{A}[\![A]\!]\langle \rho, D\rangle, \langle \rho, D\rangle \in W\} \cup W'$$

$$W' \stackrel{\text{def}}{=} \left\{\langle \rho, D\rangle \in \mathcal{E} \times \mathcal{D} \;\middle|\; \begin{array}{c}\langle v, D\rangle = \mathcal{A}[\![A]\!]\langle \rho, D\rangle, v > 0, \\ \langle \rho, D\rangle \in \mathcal{S}[\![S; \textbf{for } {}^{l}A - 1 \textbf{ do } S \textbf{ od}]\!] W\end{array}\right\}$$

$$\mathcal{S}\left[\!\left[\textbf{while } {}^{l}B \textbf{ do } S \textbf{ od}\right]\!\right] W \stackrel{\text{def}}{=} \operatorname{lfp} F$$

$$F(Y) \stackrel{\text{def}}{=} \{\langle \rho, D\rangle \in \mathcal{E} \times \mathcal{D} \mid \mathsf{ff} = \mathcal{B}[\![B]\!]\rho, \langle \rho, D\rangle \in W\} \cup W'$$

$$W' \stackrel{\text{def}}{=} \{\langle \rho, D\rangle \in \mathcal{E} \times \mathcal{D} \mid \mathsf{tt} = \mathcal{B}[\![B]\!]\rho, \langle \rho, D\rangle \in \mathcal{S}[\![S]\!] Y\}$$

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

Fig. 3: Input-Aware Concrete Semantics of Instructions

in the given environment and given the data read from the file (if any), and the (rest of) the data file (in $\mathcal{D}$) after the data is consumed.

Example 1. Let ρ be an environment that maps the variable *gpa* to the value 3.0, and let $D = \begin{bmatrix} 4.0 \\ 1.0 \\ 3.0 \end{bmatrix}$ be a data file containing three data records. We consider the expression $gpa + \textbf{input}()$, which simplifies the right-hand side of the assignment at line 9 in Figure 1. Its semantics is $\mathcal{A}[\![gpa + \textbf{input}()]\!] = \left(7.0, \begin{bmatrix} 1.0 \\ 3.0 \end{bmatrix}\right)$. ∎

We also define the standard input-agnostic semantics $\mathcal{A}'[\![A]\!] : \mathcal{E} \rightarrow \mathcal{P}(\mathbb{V})$ mapping an environment to the set of all possible values of the expression in the environment: $\mathcal{A}'[\![A]\!]\rho \stackrel{\text{def}}{=} \{v \in \mathbb{V} \mid \exists D \in \mathcal{D} \colon \langle v, _\rangle = \mathcal{A}[\![A]\!]\langle \rho, D\rangle\}$.

Similarly, the semantics of a boolean expression $\mathcal{B}[\![B]\!] : \mathcal{E} \rightarrow \{\mathsf{tt}, \mathsf{ff}\}$ maps an environment to the truth value of the expression in the given environment.

The semantics of programs $\Delta[\![P]\!] : \mathcal{L} \rightarrow \mathcal{P}(\mathcal{E} \times \mathcal{D})$ maps each program label $l \in \mathcal{L}$ to the set of all pairs of environments that are possible when the program execution is at that label, and input data files that the program can *fully read without errors* starting *from* that label. We define this semantics *backwards*, starting from the final program label where all environments in $\mathcal{E}$ are possible but only the empty data file ϵ can be read from that program label:

$$\Delta[\![P]\!] = \Delta\left[\!\left[S^{l}\right]\!\right] \stackrel{\text{def}}{=} \Delta[\![S]\!]\left(\lambda p. \begin{cases} \mathcal{E} \times \{\epsilon\} & p = l \\ \text{undefined} & \text{otherwise} \end{cases}\right) \tag{2.2}$$

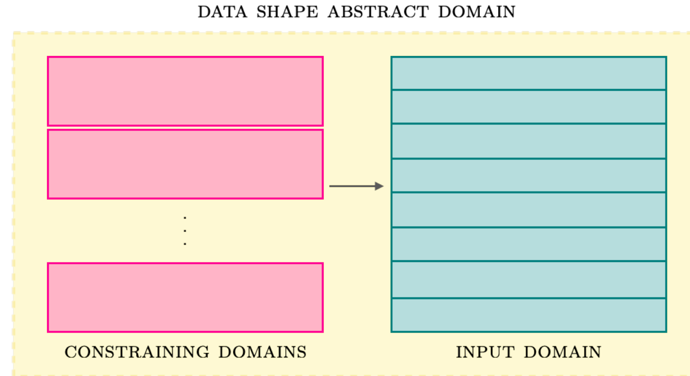

Fig. 4: Data Shape Abstract Domain.

In Figure 3, we (equivalently) define the semantics $\Delta [\![S]\!] : (\mathcal{L} \to \mathcal{P}(\mathcal{E} \times \mathcal{D})) \to (\mathcal{L} \to \mathcal{P}(\mathcal{E} \times \mathcal{D}))$ of each instruction pointwise within $\mathcal{P}(\mathcal{E} \times \mathcal{D})$: each function $\mathcal{S} [\![S]\!] : \mathcal{P}(\mathcal{E} \times \mathcal{D}) \to \mathcal{P}(\mathcal{E} \times \mathcal{D})$ takes as input a set W of pairs of environments and data files and outputs the pairs of possible environments and data files that can be read from the program label within the instruction S.

Example 2. Let ρ' be an environment that maps the variable *gpa* to the value 7.0, and let $R = \begin{bmatrix} 1.0 \\ 3.0 \end{bmatrix}$ be a data file. We consider the assignment $gpa := gpa + \textbf{input}()$ which simplifies the assignment at line 9 in Figure 1. Its semantics, given $W = \{\langle \rho', R \rangle\}$, is $\mathcal{S} [\![gpa := gpa + \textbf{input}()]\!] W = \{\langle \rho, D \rangle\}$ where ρ maps the variable *gpa* to the value 3.0 and $D = \begin{bmatrix} 4.0 \\ 1.0 \\ 3.0 \end{bmatrix}$ (see Example 1). ■

Data Shape Abstraction. In the following sections, we design a decidable abstraction of $\Delta [\![P]\!]$ which *over-approximates* the concrete semantics of P at each program label $l \in \mathcal{L}$. As a consequence, this abstraction yields *necessary preconditions* for a program to execute successfully and correctly. In particular, if a data file 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 is in the abstraction there is no guarantee that the program will execute successfully and correctly when reading data from it.

We derive the abstraction $\Delta^\natural [\![P]\!] : \mathcal{L} \to \mathcal{Q}$ by *abstract interpretation* [12]. No approximation is made on $\mathcal{L}$. On the other hand, each program label $l \in \mathcal{L}$ is associated to an element $Q \in \mathcal{Q}$ of the *data shape abstract domain* $\mathbb{Q}$. Q over-approximates the possible environments and data files read starting from l.

An overview of the data shape abstract domain is given in Figure 4. It is parameterized by a family $\mathbb{K}_1, \ldots, \mathbb{K}_k$ of *constraining abstract domains*, which collect constraints on the program variables, and an *input abstract domain* $\mathbb{H}$,

which collects constraints on the input data read by the program. We now present and describe instances of these abstract domains, before defining $\Delta^{\natural} [\![P]\!]$.

3 Constraining Abstract Domains

The constraining abstract domains abstract the possible environments at each program label. Thus, they constrain the values of the variables of the analyzed program and also *indirectly* constraint the input data read by the program.

Any constraining domain $\mathbb{K}$ that we present is characterized by a choice of:

- a set $\mathcal{K}$ of computer-representable abstract domain elements;
- a partial order $\sqsubseteq_{\mathbb{K}}$ between domain elements;
- a concretization function $\gamma_{\mathbb{K}} : \mathcal{K} \to \mathcal{P}(\mathcal{E})$ mapping abstract domain elements to sets of possible environments, or, when possible, a Galois connection $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \underset{\alpha_{\mathbb{K}}}{\overset{\gamma_{\mathbb{K}}}{\leftrightarrows}} \langle \mathcal{K}, \sqsubseteq_{\mathbb{K}} \rangle$;
- a least element $\bot_{\mathbb{K}} \in \mathcal{K}$ such that $\gamma_{\mathbb{K}}(\bot_{\mathbb{K}}) = \emptyset$;
- a greatest element $\top_{\mathbb{K}} \in \mathcal{K}$ such that $\gamma_{\mathbb{K}}(\top_{\mathbb{K}}) = \mathcal{E}$;
- a sound join operator $\sqcup_{\mathbb{K}}$ such that $\gamma_{\mathbb{K}}(K_1) \cup \gamma_{\mathbb{K}}(K_2) \subseteq \gamma_{\mathbb{K}}(K_1 \sqcup_{\mathbb{K}} K_2)$;
- a sound widening $\triangledown_{\mathbb{K}}$ if $\mathbb{K}$ does not satisfy the ascending chain condition;
- a sound backward assignment operator $\text{ASSIGN}_{\mathbb{K}} [\![X := A]\!]$ such that $\{\rho \in \mathcal{E} \mid \exists v \in \mathcal{A}' [\![A]\!] \rho \colon \rho[X \mapsto v] \in \gamma(K)\} \subseteq \gamma_{\mathbb{K}}(\text{ASSIGN}_{\mathbb{K}} [\![X := A]\!] K)$; and
- a sound filter operator $\text{FILTER}_{\mathbb{K}} [\![B]\!]$ such that $\{\rho \in \gamma_{\mathbb{K}}(K) \mid \mathsf{tt} \in \mathcal{B} [\![B]\!] \rho\} \subseteq \gamma_{\mathbb{K}}(\text{FILTER}_{\mathbb{K}} [\![B]\!] K)$.

Essentially any of the existing classical abstract domains [10,11,35, etc.] can be a constraining domain. Some of their operators just need to be augmented with certain operations to ensure the communication with the input domain $\mathbb{H}$, which (directly) constraints the input data.

Specifically, the backward assignment operation $\text{ASSIGN}_{\mathbb{K}} [\![X := A]\!]$ needs to be preceded by a $\text{REPLACE}(A, \mathcal{I})$ operation, which replaces each **input**() sub-expressions of A with a fresh special input variable $I \in \mathcal{I}$, The input variables are added to the constraining domain on the fly to track the value of the input data as well as the *order* in which the data is read by the program.

Example 3. Let us consider again the assignment $gpa := gpa +$ **input**() which simplifies line 9 in Figure 1. On way to track the order in which input data is read by the program is to parameterize the fresh input variables by the program label at which the corresponding **input**() expression occur. If we use line numbers as labels, in this case we only need one fresh input variable I_9 (for multiple **input**() expressions at the same program label 9 we can add superscripts: $I_9^1, I_9^2, \dots$). Thus, $\text{REPLACE}(gpa +$ **input**()$, \{I_9\}) = gpa + I_9$. ■

Once the assignment or filter operation has been performed, the operation $\text{RECORD}(I)$ extracts from the domain the constraints on each newly added input variable I so that they can be directly recorded in the input domain $\mathbb{H}$. The input variables can then be removed from the constraining domain $\mathbb{K}$.

3.1 Non-Relational Constraining Abstract Domains

In the following, we present a few instances of *non-relational* constraining domains. These domains abstract each program variable independently. Thus, each constraining domain element $K \in \mathcal{K}_{\mathcal{U}}$ of $\mathbb{K}_{\mathbb{U}}$ is a map $K: \mathcal{X} \to \mathcal{U}$ from program variables to elements of a *basis* abstract domain $\mathbb{U}$.

In the following, we write $\mathcal{U}[\![A]\!]K$ to denote the value (in $\mathcal{U}$) of an arithmetic expression A given the abstract domain element $K \in \mathcal{K}_{\mathcal{U}}$. In particular, for a binary expression $A_1 \diamond A_2$, we define $\mathcal{U}[\![A_1 \diamond_{\mathbb{U}} A_2]\!]K = \mathcal{U}[\![A_1]\!]K \diamond_{\mathbb{U}} \mathcal{U}[\![A_2]\!]K$ and thus we assume that the basis $\mathbb{U}$ is equipped with the operator $\diamond_{\mathbb{U}}$.

The concretization function $\gamma_{\mathbb{K}_{\mathbb{U}}}: \mathcal{K}_{\mathcal{U}} \to \mathcal{P}(\mathcal{E})$ is:

$$\gamma_{\mathbb{K}_{\mathbb{U}}}(K) \stackrel{\text{def}}{=} \{\rho \in \mathcal{E} \mid \forall X \in \mathcal{X}: \mathsf{str}(\rho(X)) \in \gamma_{\mathbb{U}}(K(X))\} \tag{3.1}$$

where $\gamma_{\mathbb{U}}: \mathcal{U} \to \mathbb{S}$ and $\mathsf{str}: \mathbb{V} \to \mathbb{S}$ converts float and integer values to strings such that $\mathsf{str}(\mathbb{F}) = \mathbb{S}_{\mathsf{float}}$ and $\mathsf{str}(\mathbb{Z}) = \mathbb{S}_{\mathsf{int}}$. The partial order $\sqsubseteq_{\mathbb{K}}$, join $\sqcup_{\mathbb{K}}$, and widening $\triangledown_{\mathbb{K}}$ are straightforwardly defined pointwise.

For these constraining domains, the $\textsc{replace}(A, \mathcal{I})$ operation *temporarily* enlarges the domain of the current abstract element $K \in \mathcal{K}_{\mathcal{U}}$ to also include input variables, i.e., $K: \mathcal{X} \cup \mathcal{I} \to \mathcal{U}$. The $\textsc{record}(I)$ operation simply returns the value $K(I) \in \mathcal{U}$. All input variable are then removed from the domain of K.

Type Constraining Abstract Domain. The first instance that we consider is very simple but interesting to catch exceptions that would be raised when casting inputs to integers or floats, as at lines 2 and 5 in Figure 1.

We define the basis type domain $\mathbb{T}$, to track *the type of input data that can be stored in the program variables.* Its elements belong to the type lattice $\mathcal{T}$ represented by the Hasse diagram in Figure 5. $\mathcal{T}$ defines the type hierarchy (reminiscent of that of PYTHON) that we use for our analysis. Data is always read as a string (cf. Section 2). Thus, STRING is the highest type in the hierarchy. Some (but not all) strings can be cast to float or integer, thus the FLOAT and INT types follow in the hierarchy. Finally, $\bot_{\mathbb{T}}$ indicates an exception.

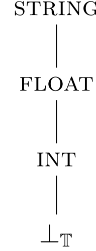

Fig. 5: The $\mathcal{T}$ type lattice.

We define the concretization function $\gamma_{\mathbb{T}}: \mathcal{T} \to \mathbb{S}$ as follows:

$$\gamma_{\mathbb{T}}(\textsc{string}) \stackrel{\text{def}}{=} \mathbb{S} \quad \gamma_{\mathbb{T}}(\textsc{float}) \stackrel{\text{def}}{=} \mathbb{S}_{\mathsf{float}} \quad \gamma_{\mathbb{T}}(\textsc{int}) \stackrel{\text{def}}{=} \mathbb{S}_{\mathsf{int}} \quad \gamma_{\mathbb{T}}(\bot_{\mathbb{T}}) \stackrel{\text{def}}{=} \emptyset \tag{3.2}$$

The partial order $\sqsubseteq_{\mathbb{T}}$, join $\sqcup_{\mathbb{T}}$, and meet $\sqcap_{\mathbb{T}}$ are defined by Figure 5. No widening $\triangledown_{\mathbb{T}}$ is necessary since the basis type domain $\mathbb{T}$ is finite.

Each element $K \in \mathcal{K}_{\mathcal{T}}$ of the type constraining abstract domain $\mathbb{K}_{\mathbb{T}}$ is thus a map $K: \mathcal{X} \to \mathcal{T}$ from program variables to type elements. The bottom element is the constant map $\lambda X.\bot_{\mathbb{T}}$ which represent a program exception. The top element

is $\lambda X.\text{STRING}$ or, better, $\lambda X.\mathsf{type}(X)$, where $\mathsf{type}(X)$ is the type inferred for X by a static type inference previously run on the program (e.g., [28,36] for PYTHON). In the latter case, the analysis with $\mathbb{K}_\mathbb{T}$ might refine the inferred type (e.g., $\mathsf{type}(X) = \text{FLOAT}$ but the analysis finds $K(X) = \text{INT}$). In particular, such a refinement is done by the $\text{ASSIGN}_{\mathbb{K}_\mathbb{T}}[\![X := A]\!]$ and $\text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![B]\!]$ operators.

The $\text{ASSIGN}_{\mathbb{K}_\mathbb{T}}[\![X := A]\!]$ operator refines the type of input data mapped to from the variables that appear in the assigned expression A. Specifically, $\text{ASSIGN}_{\mathbb{K}_\mathbb{T}}[\![X := A]\!]K \stackrel{\text{def}}{=} \text{REFINE}_{\text{REPLACE}(A,\mathcal{I})}(K[X \mapsto \mathsf{type}(X)], K(X))$, where the $\text{REFINE}_A \colon \mathcal{K} \to \mathcal{T} \to \mathcal{K}$ function is defined as follows:

$$
\begin{aligned}
\text{REFINE}_X(K,T) &\stackrel{\text{def}}{=} K[X \mapsto K(X) \sqcap_\mathbb{T} T] && X \in \mathcal{X}\\
\text{REFINE}_v(K,T) &\stackrel{\text{def}}{=} K && v \in \mathbb{V}\\
\text{REFINE}_I(K,T) &\stackrel{\text{def}}{=} K[I \mapsto T] && I \in \mathcal{I}\\
\text{REFINE}_{\mathbf{int}(A)}(K,T) &\stackrel{\text{def}}{=} \text{REFINE}_A(K, T \sqcap_\mathbb{T} \text{INT})\\
\text{REFINE}_{\mathbf{float}(A)}(K,T) &\stackrel{\text{def}}{=} \text{REFINE}_A(K, T \sqcap_\mathbb{T} \text{FLOAT})\\
\text{REFINE}_{A_1 \diamond A_2}(K,T) &\stackrel{\text{def}}{=} \text{REFINE}_{A_1}(\text{REFINE}_{A_2}(K,T'),T') && T' = T \sqcap_\mathbb{T} \text{FLOAT}
\end{aligned}
$$

Note that, for soundness, the current value $K(X)$ of the assigned variable X must be forgotten before the refinement (i.e., $K[X \mapsto \mathsf{type}(X)]$). We refine variables within an arithmetic operation $A_1 \diamond A_2$ to contain data of at most type FLOAT.

Example 4 (continue from Example 3). Let us consider again the assignment $gpa := gpa + \mathbf{input}()$ which simplifies line 9 in Figure 1 and let K be an abstract domain element which maps the variable gpa to the type value INT, while a previously ran type inference has determined that $\mathsf{type}(gpa) = \text{FLOAT}$. We have:

$$
\begin{aligned}
&\text{ASSIGN}_{\mathbb{K}_\mathbb{T}}[\![gpa := gpa + \mathbf{input}()]\!]K \stackrel{\text{def}}{=} \text{REFINE}_{gpa+I_9}(K[gpa \mapsto \text{FLOAT}], \text{INT})\\
&= \text{REFINE}_{gpa}(\text{REFINE}_{I_9}(K[gpa \mapsto \text{FLOAT}], \text{INT}), \text{INT})\\
&= \text{REFINE}_{gpa}(K[gpa \mapsto \text{FLOAT}][I_9 \mapsto \text{INT}], \text{INT}) = K[I_9 \mapsto \text{INT}][gpa \mapsto \text{INT}]
\end{aligned}
$$

which indicates that the program expects to read an integer at line 9. Note that, this is a result of our choice for K. Indeed, with K mapping gpa to FLOAT, we have $\text{ASSIGN}_{\mathbb{K}_\mathbb{T}}[\![gpa := gpa + \mathbf{input}()]\!]K = K[I_9 \mapsto \text{FLOAT}][gpa \mapsto \text{FLOAT}]$ (which is what the program in Figure 1 actually expects). ■

Similarly, the filter operator $\text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![B]\!]$ is defined as follows:

$$
\begin{aligned}
\text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![A_1 = A_2]\!]K &\stackrel{\text{def}}{=} \text{REFINE}_{A_1}(\text{REFINE}_{A_2}(K, \mathcal{T}[\![A_1]\!]K), \mathcal{T}[\![A_2]\!]K)\\
\text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![A_1 \neq A_2]\!]K &\stackrel{\text{def}}{=} K\\
\text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![A_1 \,\overline{\bowtie}\, A_2]\!]K &\stackrel{\text{def}}{=} \text{REFINE}_{A_1}(\text{REFINE}_{A_2}(K, \text{FLOAT}), \text{FLOAT})\\
\text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![B_1 \vee B_2]\!]K &\stackrel{\text{def}}{=} \text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![B_1]\!]K \sqcup_{\mathbb{K}_\mathbb{N}} \text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![B_2]\!]K\\
\text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![B_1 \wedge B_2]\!]K &\stackrel{\text{def}}{=} \text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![B_2]\!]K \circ \text{FILTER}_{\mathbb{K}_\mathbb{T}}[\![B_1]\!]K
\end{aligned}
$$

where $\overline{\bowtie} \in \{<, \leq, >, \geq\}$.

The soundness of the domain operators is straightforward:

Lemma 1. *The operators of the type constraining domain $\mathbb{K}_\mathbb{T}$ are sound.*

Value Constraining Abstract Domains. Numerical abstract domains such as the interval domain [11] or the sign domain [13] can be used to track *the input data values that can be stored in the program variables*. In particular, the latter is useful to catch exceptions raised when diving by zero, as at line 10 in Figure 1.

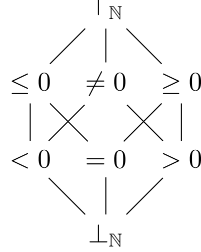

Fig. 6: The $\mathcal{N}$ sign lattice.

The sign lattice $\mathcal{N}$ shown in Figure 6 represents the elements of the basis sign domain $\mathbb{N}$. We define the concretization function $\gamma_{\mathbb{N}} \colon \mathcal{N} \to \mathbb{S}$ as follows:

$$\begin{aligned} \gamma_{\mathbb{N}}(\top_{\mathbb{N}}) &\stackrel{\text{def}}{=} \mathbb{S} \\ \gamma_{\mathbb{N}}(\lhd 0) &\stackrel{\text{def}}{=} \mathbb{S}^{\lhd 0}_{\mathsf{float}} \\ \gamma_{\mathbb{N}}(\bot_{\mathbb{N}}) &\stackrel{\text{def}}{=} \emptyset \end{aligned} \tag{3.3}$$

where $\lhd \in \{<, \leq, =, \neq, >, \geq\}$ and $\mathbb{S}^{\lhd 0}_{\mathsf{float}}$ denotes the set of string values that can be interpreted as float values that satisfy $\lhd 0$. The partial order $\sqsubseteq_{\mathbb{N}}$, join $\sqcup_{\mathbb{N}}$, and meet $\sqcap_{\mathbb{N}}$ are defined by the Hasse diagram in Figure 6. Again, no widening $\triangledown_{\mathbb{N}}$ is necessary since the basis domain $\mathbb{N}$ is finite.

Each element $K \in \mathcal{K}_{\mathcal{N}}$ of the sign constraining abstract domain $\mathbb{K}_{\mathbb{N}}$ is thus a map $K \colon \mathcal{X} \to \mathcal{N}$ from program variables to sign elements.

For this domain, the backward assignment operator is $\textsc{assign}_{\mathbb{K}_{\mathbb{N}}} [\![X := A]\!] K \stackrel{\text{def}}{=} \textsc{refine}_{\textsc{replace}(A,\mathcal{I})}(K[X \mapsto \top_{\mathbb{N}}], K(X))$, where $\textsc{refine}_A \colon \mathcal{K} \to \mathcal{N} \to \mathcal{K}$ is:

$$\begin{aligned}
\textsc{refine}_X(K, N) &\stackrel{\text{def}}{=} K[X \mapsto K(X) \sqcap_{\mathbb{N}} N] && X \in \mathcal{X} \\
\textsc{refine}_v(K, N) &\stackrel{\text{def}}{=} K && v \in \mathbb{V} \\
\textsc{refine}_I(K, N) &\stackrel{\text{def}}{=} K[I \mapsto N] && I \in \mathcal{I} \\
\textsc{refine}_{\mathbf{int}(A)}(K, N) &= \textsc{refine}_{\mathbf{float}(A)}(K, N) \stackrel{\text{def}}{=} \textsc{refine}_A(K, N) \\
\textsc{refine}_{A_1 + A_2}(K, N) &\stackrel{\text{def}}{=} \textsc{refine}_{A_1}(\textsc{refine}_{A_2}(K, N -_{\mathbb{N}} \mathcal{N}[\![A_1]\!] K), N -_{\mathbb{N}} \mathcal{N}[\![A_2]\!] K) \\
\textsc{refine}_{A_1 - A_2}(K, N) &\stackrel{\text{def}}{=} \textsc{refine}_{A_1}(\textsc{refine}_{A_2}(K, \mathcal{N}[\![A_1]\!] K -_{\mathbb{N}} N), N +_{\mathbb{N}} \mathcal{N}[\![A_2]\!] K) \\
\textsc{refine}_{A_1 * A_2}(K, N) &\stackrel{\text{def}}{=} \textsc{refine}_{A_1}(\textsc{refine}_{A_2}(K, N /_{\mathbb{N}} \mathcal{N}[\![A_1]\!] K), N /_{\mathbb{N}} \mathcal{N}[\![A_2]\!] K) \\
\textsc{refine}_{A_1 / A_2}(K, N) &\stackrel{\text{def}}{=} \textsc{refine}_{A_1}(K', N *_{\mathbb{N}} \mathcal{N}[\![A_2]\!] K) \\
& \quad K' = \textsc{refine}_{A_2}(K, \neq 0 \sqcap_{\mathbb{N}} (\mathcal{N}[\![A_1]\!] K /_{\mathbb{N}} N))
\end{aligned}$$

Note that we refine variables in the denumerator A_2 of a division expression $A_1 \div A_2$ to have values different from zero.

Example 5. Let us consider the assignment *result* := *gpa* / *classes* at line 10 in Figure 1 and let K be an abstract domain element which maps the variables *gpa* and *result* to the sign value ≥ 0 and the variable *classes* to $\top_{\mathbb{N}}$. We have:

$$\begin{aligned}
&\textsc{assign}_{\mathbb{K}_{\mathbb{N}}} [\![\mathit{result} := \mathit{gpa} \mathbin{/} \mathit{classes}]\!] K \stackrel{\text{def}}{=} \textsc{refine}_{\mathit{gpa}/\mathit{classes}}(K[\mathit{result} \mapsto \top_{\mathbb{N}}], \geq 0) \\
&= \textsc{refine}_{\mathit{gpa}}(\textsc{refine}_{\mathit{classes}}(K[\mathit{gpa} \mapsto \top_{\mathbb{N}}], \neq 0), \geq 0) \\
&= \textsc{refine}_{\mathit{gpa}}(K[\mathit{result} \mapsto \top_{\mathbb{N}}][\mathit{classes} \mapsto \neq 0], \geq 0) \\
&= K[\mathit{result} \mapsto \top_{\mathbb{N}}][\mathit{classes} \mapsto \neq 0][\mathit{gpa} \mapsto \geq 0]
\end{aligned}$$

which, in particular, indicates that the program expects the variable *classes* (read at line 5 in Figure 1) to have a value different from zero. ■

Instead, the filter operator $\text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![B]\!]$ is defined as follows:

$$\begin{aligned}
\text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![A \lhd 0]\!]\, K &\stackrel{\text{def}}{=} \text{REFINE}_A(K, \lhd 0) && \lhd \in \{<, \leq, =, \neq, >, \geq\}\\
\text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![A_1 \bowtie A_2]\!]\, K &\stackrel{\text{def}}{=} \text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![A_1 - A_2 \bowtie 0]\!]\, K && A_2 \neq 0\\
\text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![B_1 \vee B_2]\!]\, K &\stackrel{\text{def}}{=} \text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![B_1]\!]\, K \sqcup_{\mathbb{K}_\mathbb{N}} \text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![B_2]\!]\, K\\
\text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![B_1 \wedge B_2]\!]\, K &\stackrel{\text{def}}{=} \text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![B_2]\!]\, K \circ \text{FILTER}_{\mathbb{K}_\mathbb{N}} [\![B_1]\!]\, K
\end{aligned}$$

The soundness of the sign constraining domain operators follows directly from the soundness of the sign abstract domain [13].

Lemma 2. *The operators of the sign constraining domain $\mathbb{K}_\mathbb{N}$ are sound.*

String Constraining Abstract Domains. Finally, we build a last instance of non-relational constraining domain on the finite string set domain [9], to track *the string data values that can be stored in the program variables*. Other more sophisticated string domains exist [2,10, etc.]. However, even this simple domain suffices to catch KeyError exceptions that might occur, e.g., at line 9 in Figure 1.

Each abstract domain element $K \in \mathcal{K}_\mathcal{W}$ of the string domain $\mathbb{K}_\mathbb{W}$ is a map $K : \mathcal{X} \to \mathcal{W}$ from program variables to an element $W \in \mathcal{W}$ of the basis domain $\mathbb{W}$. Elements of $\mathbb{W}$ are finite sets of at most m string, or the top element $\top_\mathbb{W}$ which abstracts larger sets of strings, i.e., $\mathcal{W} \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{S}) \cup \{\top_\mathbb{W}\}$. In the following, we write $\bot_\mathbb{W}$ to denote the empty string set. The concretization function $\gamma_\mathbb{W} : \mathcal{W} \to \mathbb{S}$ is:

$$\gamma_\mathbb{W}(\top_\mathbb{W}) \stackrel{\text{def}}{=} \mathbb{S} \qquad \gamma_\mathbb{W}(W) \stackrel{\text{def}}{=} W \tag{3.4}$$

The partial order $\sqsubseteq_\mathbb{W}$, join $\sqcup_\mathbb{W}$, and meet $\sqcap_\mathbb{W}$ are the set operations $\subseteq$, $\cup$, and $\cap$ extended to also handle $\top_\mathbb{W}$:

$$\begin{aligned}
&W_1 \sqsubseteq W_2 \Leftrightarrow W_2 = \top_\mathbb{W} \vee (W_1 \neq \top_\mathbb{W} \wedge W_1 \subseteq W_2)\\
&W_1 \sqcup_\mathbb{W} W_2 \stackrel{\text{def}}{=} \begin{cases} \top_\mathbb{W} & W_1 = \top_\mathbb{W} \vee W_2 = \top_\mathbb{W} \vee |W_1 \cup W_2| > m\\ W_1 \cup W_2 & \text{otherwise}\end{cases}\\
&W_1 \sqcap_\mathbb{W} W_2 \stackrel{\text{def}}{=} \begin{cases} W_1 & W_2 = \top_\mathbb{W}\\ W_2 & W_1 = \top_\mathbb{W}\\ W_1 \cap W_2 & \text{otherwise}\end{cases}
\end{aligned}$$

The widening $W_1 \triangledown_\mathbb{W} W_2$ yields $\top_\mathbb{W}$ unless $W_2 \subseteq W_1$ (in which case it yields W_1).

We can now define the backward assignment operator $\text{ASSIGN}_{\mathbb{K}_\mathbb{W}} [\![X := A]\!]\, K \stackrel{\text{def}}{=} \text{REFINE}_{\text{REPLACE}(A,\mathcal{I})}(K[X \mapsto \top_\mathbb{W}], K(X))$, where $\text{REFINE}_A : \mathcal{K} \to \mathcal{W} \to \mathcal{K}$ is:

$$\begin{aligned}
\text{REFINE}_X(K, W) &\stackrel{\text{def}}{=} K[X \mapsto K(X) \sqcap_\mathbb{W} W] && X \in \mathcal{X}\\
\text{REFINE}_v(K, W) &\stackrel{\text{def}}{=} K && v \in \mathbb{V}\\
\text{REFINE}_I(K, W) &\stackrel{\text{def}}{=} K[I \mapsto W] && I \in \mathcal{I}\\
\text{REFINE}_{\mathbf{int}(A)}(K, W) = \text{REFINE}_{\mathbf{float}(A)}(K, W) &\stackrel{\text{def}}{=} \text{REFINE}_A(K, W) && W = \top_\mathbb{W}\\
\text{REFINE}_{\mathbf{int}(A)}(K, W) = \text{REFINE}_{\mathbf{float}(A)}(K, W) &\stackrel{\text{def}}{=} \text{REFINE}_A(K, \bot_\mathbb{W}) && W \neq \top_\mathbb{W}\\
\text{REFINE}_{A_1 \diamond A_2}(K, W) &\stackrel{\text{def}}{=} \text{REFINE}_{A_1}(\text{REFINE}_{A_2}(K, W), W) && W = \top_\mathbb{W}\\
\text{REFINE}_{A_1 \diamond A_2}(K, W) &\stackrel{\text{def}}{=} \text{REFINE}_{A_1}(\text{REFINE}_{A_2}(K, \bot_\mathbb{W}), \bot_\mathbb{W}) && W \neq \top_\mathbb{W}
\end{aligned}$$

Note that, variables in numerical expressions (such as $\mathbf{int}(A)$, $\mathbf{float}(A)$ or $A_1 \diamond A_2$) should not have a specific string value (i.e, a value different from $\top_{\mathbb{W}}$).

Example 6. Let us consider a small extension of our toy language with dictionaries. In particular, we extend the grammar of arithmetic expressions with dictionary display (in PYTHON terminology) expressions $\{v_0 : v_1, v_2 : v_3, \dots\}$, $v_0, v_1, v_2, v_3, \dots \in \mathbb{V}$, for dictionary creation (cf. line 1 in Figure 1) and dictionary access expressions $X[A]$ (such as $grade2gpa[grade]$ at line 9 in Figure 1).

For each dictionary, we assume that abstract domains only keep track of two summary variables [25], one representing the dictionary keys and one representing its values. For instance, let us consider the $grade2gpa$ dictionary in Figure 1 and let the string domain element K map the variable $keys(grade2gpa)$ to the set of strings {'A', 'B', 'C', 'D', 'F'} and $values(grade2gpa)$ to $\top_{\mathbb{N}}$.

We can extend REFINE_A defined above to handle dictionary access expressions as follows: $\text{REFINE}_{X[A]}(K, W) \stackrel{\text{def}}{=} \text{REFINE}_A(K, K(keys(X)))$. No refinement can be made on X since, for soundness, only weak updates are allowed on summary variables [7]. For the assignment $gpa := gpa + grade2gpa[grade]$ at line 9 in Figure 1 we thus have $\text{ASSIGN}_{\mathbb{K}_{\mathbb{W}}} [\![gpa := gpa + grade2gpa[grade]]\!] K = K[grade \mapsto \{'A', 'B', 'C', 'D', 'F'\}]$, which indicates the string values expected by the program for the variable $grade$ (read at line 8 in Figure 1). ■

The filter operator $\text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![B]\!]$ is defined as follows:

$$
\begin{aligned}
\text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![A_1 = A_2]\!] K &\stackrel{\text{def}}{=} \text{REFINE}_{A_1}(\text{REFINE}_{A_2}(K, \mathcal{W} [\![A_1]\!] K), \mathcal{W} [\![A_2]\!] K) \\
\text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![A_1 \neq A_2]\!] K &\stackrel{\text{def}}{=} K \\
\text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![A_1 \,\overline{\bowtie}\, A_2]\!] K &\stackrel{\text{def}}{=} \text{REFINE}_{A_1}(\text{REFINE}_{A_2}(K, \mathcal{W} [\![A_1]\!] K), \mathcal{W} [\![A_2]\!] K) \\
&\qquad \mathcal{W} [\![A_2]\!] K = \top_{\mathbb{W}} \wedge \mathcal{W} [\![A_1]\!] K = \top_{\mathbb{W}} \\
\text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![A_1 \,\overline{\bowtie}\, A_2]\!] K &\stackrel{\text{def}}{=} \text{REFINE}_{A_1}(\text{REFINE}_{A_2}(K, \bot_{\mathbb{W}}), \bot_{\mathbb{W}}) \\
&\qquad \mathcal{W} [\![A_2]\!] K \neq \top_{\mathbb{W}} \vee \mathcal{W} [\![A_1]\!] K \neq \top_{\mathbb{W}} \\
\text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![B_1 \vee B_2]\!] K &\stackrel{\text{def}}{=} \text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![B_1]\!] K \sqcup_{\mathbb{K}_{\mathbb{W}}} \text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![B_2]\!] K \\
\text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![B_1 \wedge B_2]\!] K &\stackrel{\text{def}}{=} \text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![B_2]\!] K \circ \text{FILTER}_{\mathbb{K}_{\mathbb{W}}} [\![B_1]\!] K
\end{aligned}
$$

where $\overline{\bowtie} \in \{<, \leq, >, \geq\}$.

The soundness of the string constraining domain operators follows directly from the soundness of the finite string set abstract domain [9].

Lemma 3. *The operators of the string constraining domain $\mathbb{K}_{\mathbb{W}}$ are sound.*

3.2 Other Constraining Abstract Domains

We now briefly discuss other instances of constraining domain.

Relational Constraining Abstract Domains. Other constraining domain can be built on *relational* abstract domains. Popular such domains are octagons [35] or polyhedra [16], which track linear relations between program variables.

We refer to the literature for the formal definition of these abstract domains and only discuss here the implementation of the additional operations needed to communicate with the input domain $\mathbb{H}$. In particular, similarly to non-relational domains, the $\text{REPLACE}(E, \mathcal{I})$ operation temporarily adds the input variables in $\mathcal{I}$ to the current abstract element $K \in \mathcal{K}$. These are unconstrained at first and might become subjects to constraints after an assignment or filter operation.

The implementation of the $\text{RECORD}(I)$ operation is more complex for relational domains: $\text{RECORD}(I)$ extracts from the current abstract element K, an abstract domain element $\overline{K}$ containing all and only the constraints in K that involve the input variable I. The domain $\text{dom}(\overline{K})$ of $\overline{K}$ is the subset of $\text{dom}(K)$ containing only the variables appearing in these constraints. The input variables can then be projected away from K.

Example 7. Let us consider again the assignment $gpa := gpa + \textbf{input}()$ which simplifies line 9 in Figure 1 and let $K = \{gpa \geq 0, grades > 0\}$ be a polyhedra defined over the variables gpa and $grades$, i.e., $\text{dom}(K) = \{gpa, grades\}$. After $\text{REPLACE}(gpa + \textbf{input}(), \{I_9\})$ (cf. Example 3), K is unchanged but its domain is enlarged to also include the input variable I_9, i.e., $\text{dom}(K) = \{gpa, grades, I_9\}$. The result of the (replaced) assignment $gpa := gpa + I_9$ is then the polyhedra $K' = \{gpa + I_9 \geq 0, grades > 0\}$. Finally, the $\text{RECORD}(I_9)$ operation returns the polyhedra $\overline{K} = \{gpa + I_9 \geq 0\}$, where $\text{dom}(\overline{K}) = \{gpa, I_9\}$. ∎

In the following, we assume that input variables are parameterized by the program label at which their corresponding **input**() expressions occur, as in Example 3. Note that, there is not necessarily a one-to-one correspondence between **input**() expressions in a program and data record in a data file. Indeed, multiple records can be read by the same **input**() expression (i.e., in a for loop as in Figure 1) or, vice versa, the same data record could be read by multiple **input**() expressions (i.e., in different if branches). In particular, the latter case implies that two abstract domain elements K_1 and K_2 might be defined over different input variables. Thus, relational constraining domains need to be equipped with a unification operation $\text{UNIFY}(K_1, K_2)$ to match different input variables that correspond to the same data record. One simple option to deal with this problem is to keep track of the order in which the input variables are added to a domain element by each $\text{REPLACE}(E, \mathcal{I})$ operation. The $\text{UNIFY}(K_1, K_2)$ operation then simply consists in matching input variables in their order.

Container Constraining Abstract Domains. We now lift the assumption that data records only have one field (cf. Section 2). We extend the grammar of expressions to also include data access expressions $X[A]$, $X \in \mathcal{X}$, (similarly to what we did in Example 6 for dictionaries). Similarly, we extend the grammar of statements to also include assignments of the form $X[A_1] := A_2$. We call variables like the X we used in these expressions, *array variables*.

In this case, abstract domains should be able to also handle reads and updates of array variables in addition to numerical and string variables as so far. The most basic option to do so is to use summarization [25] (as in Example 6) and only

perform weak updates [7]. It is sometimes possible to fully expand array variables to improve precision [4], or use a combination of expansion and summarization (i.e., expand part of the array up to a certain size and summarize the rest).

Many other abstract domains exist that are specifically designed to analyze arrays [15,26,27,33, etc.] or, more generally, containers (e.g., sets, dictionaries) [18,17,20,21,22, etc.]. Any of these can be instantiated as a constraining domain (as we showed in this section) and used within our framework.

4 Input Abstract Domain

The input abstract domain $\mathbb{H}$, as mentioned, *directly* constrains the input data read by a program. An element $H \in \mathcal{H}$ of $\mathbb{H}$ is a *stack* of mutable length h:

$$R_0 \mid R_1 \mid \cdots \mid R_{h-1} \mid R_h \qquad\qquad R_i \in \mathcal{R}$$

of assumptions on (part of) the input data, or the special element $\bot_\mathbb{Q}$ or $\top_\mathbb{Q}$. The top element $\top_\mathbb{Q}$ denotes unconstrained input data, while $\bot_\mathbb{Q}$ indicates a program exception. A stack element grows or shrinks based on the level of nesting of the currently analyzed **input**() expression.

Each layer $R_i \in \mathcal{R}$ is a list of r assumptions repeated M times: $\mathcal{R} \stackrel{\text{def}}{=} \{M \cdot (J_i)_{i=1}^r \mid J_i \in \mathcal{C} \cup \{\bigstar\} \cup \mathcal{R}\}$. The *multiplier* M follows this grammar:

$$M ::= X \in \mathcal{X} \mid I \in \mathcal{I} \mid v \in \mathbb{Z} \mid M_1 \diamond M_2 \qquad\qquad \diamond \in \{+, -, *, /\}$$

while an assumption J_i can be a *basic assumption* in $\mathcal{C}$, the *dummy assumption* $\bigstar$, or another list of repeated assumptions in $\mathcal{R}$.

A basic assumption $C \in \mathcal{C}$ is a family of constraints, one for each constraining domain $\mathbb{K}_1, \dots, \mathbb{K}_k$ in $\mathbb{Q}$, associated to a particular program label $l \in \mathcal{L}$: $\mathcal{C} \stackrel{\text{def}}{=} \left\{\langle l, (Y_i)_{i=1}^k \rangle \mid l \in \mathcal{L}, Y_i \in \overline{\mathcal{K}_i}\right\}$, where $\overline{\mathcal{K}_i} = \mathcal{U}_i$ if $\mathbb{K}_i$ is a non-relational domain, or $\mathcal{K}_i$ otherwise (cf. Section 3).

Example 8. Let us consider the assignment *grade* := I_8 where I_8 is the result of REPLACE(**input**(), $\{I_8\}$) at line 8 in Figure 1. Moreover, let $K_T \in \mathbb{K}_\mathbb{T}$ and $K_W \in \mathbb{K}_\mathbb{W}$ map the variable *grade* to STRING and {'A', 'B', 'C', 'D', 'F'}, respectively. After the analysis of the assignment, we have $K_T(I_8) =$ STRING and $K_W(I_8) =$ {'A', 'B', 'C', 'D', 'F'}. The call to the function RECORD(I_8) in the two constraining domains $\mathbb{K}_\mathbb{T}$ and $\mathbb{K}_\mathbb{W}$ effectively creates the basic assumption $\langle l_8$, [STRING, {'A', 'B', 'C', 'D', 'F'}]$\rangle$ in the input domain $\mathbb{H}$. ∎

A repeated assumption $M \cdot (J_i)_{i=1}^r$ constrains *all* data read by a for loop.

Example 9 (continue from Example 8). Let us consider the for loop at lines 7-9 in Figure 1. The **input**() expression at line 8 is constrained by the basic assumption $\langle l_8$, [STRING, {'A', 'B', 'C', 'D', 'F'}]$\rangle$. Thus, all data read by the for loop is constrained by *classes* · [$\langle l_8$, [STRING, {'A', 'B', 'C', 'D', 'F'}]$\rangle$]. ∎

Finally, data read by a while loop is generally approximated by the dummy assumption $\bigstar$, which denotes an unknown number of unconstrained data records.

The concretization function $\gamma_{\mathbb{H}} \colon \mathcal{H} \to \mathcal{P}(\mathcal{D})$ is defined as follows:

$$\begin{aligned} \gamma_{\mathbb{H}}(\bot_{\mathbb{H}}) &\stackrel{\text{def}}{=} \emptyset \\ \gamma_{\mathbb{H}}(H) &\stackrel{\text{def}}{=} \{D \in \mathcal{D} \mid D \models H\} \\ \gamma_{\mathbb{H}}(\top_{\mathbb{H}}) &\stackrel{\text{def}}{=} \mathcal{D} \end{aligned} \tag{4.1}$$

In particular, the concretization of a stack element $H \in \mathbb{H}$ is the set of data files that *satisfy* the assumptions fixed by the stack element. We omit the formal definition of the satisfaction relation $\models$ due to space limitations. The following example should provide an intuition:

Example 10. Let us assume that the program in Figure 1 is analyzed with $\mathbb{Q}$ instantiated with the type $\mathbb{K}_{\mathbb{T}}$, sign $\mathbb{K}_{\mathbb{N}}$, and string $\mathbb{K}_{\mathbb{W}}$ constraining domains. Let us consider the following stack element $H \in \mathbb{H}$ at line 5:

$$1 \cdot [\langle l_5, [\text{INT}, \neq 0, \top_{\mathbb{W}}] \rangle, I_5 \cdot [\langle l_8, [\text{STRING}, \top_{\mathbb{N}}, \{\text{'A'}, \text{'B'}, \text{'C'}, \text{'D'}, \text{'F'}\}] \rangle]] \mid 1 \cdot []$$

The data file $\begin{bmatrix} 2 \\ \text{A} \\ \text{F} \end{bmatrix}$ satisfies H since $2 \in \gamma_{\mathbb{T}}(\text{INT}) \cap \gamma_{\mathbb{N}}(\neq 0) \cap \gamma_{\mathbb{W}}(\top_{\mathbb{W}})$ and, similarly, $\text{A}, \text{F} \in \gamma_{\mathbb{T}}(\text{STRING}) \cap \gamma_{\mathbb{N}}(\top_{\mathbb{N}}) \cap \gamma_{\mathbb{W}}(\{\text{'A'}, \text{'B'}, \text{'C'}, \text{'D'}, \text{'F'}\})$. Moreover, $I_5 = 2$ and, indeed, there are exactly two data records following 2. Instead, the data file $\begin{bmatrix} 1 \\ \text{A} \\ \text{F} \end{bmatrix}$ (cf. the motivating example in the Introduction) does not satisfy H since $I_5 = 1$ is followed by two data records instead of one. ∎

Any data file satisfies the dummy assumption $\bigstar$. Thus any stack element starting with the dummy assumption (e.g, $1 \cdot [\bigstar]$) is equivalent to $\top_{\mathbb{H}}$.

We define the partial order $\sqsubseteq_{\mathbb{H}}$ such that $H_1 \sqsubseteq_{\mathbb{H}} H_2$ only if $\gamma_{\mathbb{H}}(H_1) \subseteq \gamma_{\mathbb{H}}(H_2)$. Thus, $H_1 \sqsubseteq_{\mathbb{H}} H_2$ is always true if $H_1 = \bot_{\mathbb{H}}$ or $H_2 = \top_{\mathbb{H}}$. Otherwise, H_1 and H_2 must have the same number of layers to be comparable and $\sqsubseteq_{\mathbb{H}}$ is defined laywer-wise. Specifically, for each $R_1 = M_1 \cdot \left[J_1^1, \ldots, J_{r_1}^1\right] \in \mathcal{R}$ and $R_2 = M_2 \cdot \left[J_1^2, \ldots, J_{r_2}^2\right] \in \mathcal{R}$, $R_1 \sqsubseteq_{\mathbb{R}} R_2$ if and only if $M_1 = M_2$ and $r_1 = r_2$ (i.e., R_1 and R_2 consists of the same number of assumptions repeated the same number of times), and $\bigwedge_{i=1}^{r_1=r_2} J_i^1 \sqsubseteq_{\mathbb{J}} J_i^2$, i.e., R_1 imposes stronger constraints on the input data than R_2. The partial order $J_1 \sqsubseteq_{\mathbb{J}} J_2$ is again $J_1 \sqsubseteq_{\mathbb{R}}$, if $J_1, J_2 \in \mathcal{R}$. Otherwise, $J_1 \sqsubseteq_{\mathbb{J}} J_2$ is always true when $J_2 = \bigstar$. For basic assumptions $J_1 = \langle l_1, \left[Y_0^1, \ldots, Y_k^1\right] \rangle \in \mathcal{C}$ and $J_2 = \langle l_1, \left[Y_0^2, \ldots, Y_k^2\right] \rangle \in \mathcal{C}$, $J_1 \sqsubseteq_{\mathbb{J}} J_2$ is true if and only if $\bigwedge_{i_1}^{k} Y^1 \sqsubseteq_{\overline{\mathbb{K}}_i} Y^2$, where $\overline{\mathbb{K}}_i = \mathbb{U}_i$ if $\mathbb{K}_i$ is a non-relational domain, or $\mathbb{K}_i$ otherwise. Note that, for relational domains, a unification must be performed prior to $\sqsubseteq_{\mathbb{J}}$ as discussed in Section 3. No comparison is possible when $J_1 \in \mathcal{C}$ and $J_2 \in \mathcal{R}$, or vice versa.

This is a rather rigid definition for $\sqsubseteq_{\mathbb{H}}$. Indeed, in some cases, $H_1 \not\sqsubseteq_{\mathbb{H}} H_2$ even though $\gamma_{\mathbb{H}}(H_1) \subseteq \gamma_{\mathbb{H}}(H_2)$, e.g., consider $H_1 = 1 \cdot [\langle l_a, [\text{FLOAT}] \rangle, \langle l_b, [\text{FLOAT}] \rangle]$

and $H_2 = 2 \cdot [\langle l_c, [\text{FLOAT}]\rangle]$. Such incomparable stack elements may result from syntactically different but semantically close programs [19] (e.g., in one program a loop has been unrolled but not in the other), but never during the analysis of a single program. Thus, for our purposes, this definition of $\sqsubseteq_{\mathbb{H}}$ suffices.

The join $\sqcup_{\mathbb{H}}$ is defined analogously to $\sqsubseteq_{\mathbb{H}}$. We omit its formal definition due to space limitations. The join of incomparable stack layers is approximated with the dummy layer $1 \cdot [\bigstar]$. Thus, no widening $\triangledown_{\mathbb{H}}$ is needed.

The backward assignment operator $\text{ASSIGN}_{\mathbb{H}} [\![X := A]\!]$ and filter operator $\text{FILTER}_{\mathbb{H}} [\![B]\!]$ operate on each stack layer independently. For each $R = M \cdot (J_i)_{i=1}^{r} \in \mathcal{R}$, the assignment replaces any occurrence of X in the multiplier M with the expression $\text{REPLACE}(A, \mathcal{I})$. The assignment (resp. filter) operation is done recursively on each assumption J_i. When $J_i \in \mathcal{C}$, the assignment (resp. filter) is delegated to the constraining domains directly.

Example 11 (continue from Example 9). Let us consider again the assumption $classes \cdot [\langle l_8, [\text{STRING}, \{\text{'A', 'B', 'C', 'D', 'F'}\}]\rangle]$, which constrains the data read by the for loop at lines 7-9 in Figure 1, and the assignment $classes :=$ **input**() (cf. line 5). The assignment simply replaces the multiplier *classes* in the assumption with the input variable I_5: $I_5 \cdot [\langle l_8, [\text{STRING}, \{\text{'A', 'B', 'C', 'D', 'F'}\}]\rangle]$. ■

During the analysis of a for loop, the $\text{REPEAT} [\![A]\!]$ operator modifies the multiplier of the assumption in the first stack layer: $\text{REPEAT} [\![A]\!] (M \cdot [J, \dots] \mid \cdots \mid R_h) \stackrel{\text{def}}{=} (A * M) \cdot [J, \dots] \mid \cdots \mid R_h$. The resulting multiplier expression is then simplified, whenever possible (e.g, $(X + 1) - 1$ is simplified to X).

Finally, it remains to discuss how stack elements $H \in \mathcal{H}$ grow and shrink during the analysis of a program. Whenever the analysis enters the body of an if or loop statement, the $\text{PUSH}(H)$ operation simply adds an extra layer to H containing the empty assumption $1 \cdot []$: $\text{PUSH}(H) \stackrel{\text{def}}{=} 1 \cdot [] \mid H$. When the analysis later leaves the body of the statement, the $\text{POP}(H)$ operation inserts the assumption in the first layer into the assumption in the second layer: $\text{POP}(R_0 \mid M \cdot [J, \dots] \mid \cdots \mid R_h) = M \cdot [R_0, J, \dots] \mid \cdots \mid R_h$. Instead, the $\overline{\text{POP}}$ operation merges the assumption in the first layer with the (first) assumption in the second layer: $\text{POP}(R_0 \mid M \cdot [J, \dots] \mid \cdots \mid R_h) = M \cdot [R_0 \sqcup_{\mathbb{J}} J, \dots] \mid \cdots \mid R_h$.

The input domain operators ultimately build on the operators of the constraining domains. Thus, their soundness directly follows from that of the constraining domain operators.

Lemma 4. *The operators of the input domain $\mathbb{H}$ are sound.*

5 Input Data-Aware Program Abstraction

We can now use the data shape abstract domain $\mathbb{Q}$ to define the abstract semantics $\Delta^{\natural} [\![P]\!]$. We write $\langle \langle K_1, \dots, K_k \rangle, H \rangle \in \mathcal{Q}$ to denote an element of $\mathbb{Q}$, where $K_1 \in \mathcal{K}_1, \dots, K_k \in \mathcal{K}_k$ are elements of the constraining domains $\mathbb{K}_1, \dots, \mathbb{K}_k$ and

$$\mathcal{S}^\natural \left[\!\left[{}^l X := A \right]\!\right] Q \stackrel{\text{def}}{=} \text{ASSIGN}_{\mathbb{Q}} [\![X := A]\!] Q$$

$$\mathcal{S}^\natural \left[\!\left[\textbf{if } {}^l B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \right]\!\right] Q \stackrel{\text{def}}{=} Q_1 \sqcup_{\mathbb{Q}} Q_2$$

$$Q_1 \stackrel{\text{def}}{=} \text{POP} \circ \text{FILTER}_{\mathbb{Q}} [\![B]\!] \circ \mathcal{S}^\natural [\![S_1]\!] \circ \text{PUSH}(Q)$$

$$Q_2 \stackrel{\text{def}}{=} \text{POP} \circ \text{FILTER}_{\mathbb{Q}} [\![\neg B]\!] \circ \mathcal{S}^\natural [\![S_2]\!] \circ \text{PUSH}(Q)$$

$$\mathcal{S}^\natural \left[\!\left[\textbf{for } {}^l A \textbf{ do } S \textbf{ od} \right]\!\right] Q \stackrel{\text{def}}{=} \text{lfp}^\natural_{\text{POP}\circ\text{REPEAT}[\![A]\!]\circ\mathcal{S}^\natural[\![S]\!]\circ\text{PUSH}(W)}\ G$$

$$G(Y) \stackrel{\text{def}}{=} \overline{\text{POP}} \circ \text{REPEAT} [\![A]\!] \circ \mathcal{S}^\natural [\![S]\!] \circ \text{PUSH}(Y)$$

$$\mathcal{S}^\natural \left[\!\left[\textbf{while } {}^l B \textbf{ do } S \textbf{ od} \right]\!\right] Q \stackrel{\text{def}}{=} \text{lfp}^\natural\ F$$

$$F(Y) \stackrel{\text{def}}{=} \text{POP} \circ \text{FILTER}_{\mathbb{Q}} [\![\neg B]\!] \circ \text{PUSH}(Q) \sqcup_{\mathbb{Q}} \text{POP} \circ \text{FILTER}_{\mathbb{Q}} [\![B]\!] \circ \mathcal{S}^\natural [\![S]\!] \circ \text{PUSH}(Y)$$

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

Fig. 7: Input-Aware Abstract Semantics of Instructions

$H \in \mathcal{H}$ is an element of the input domain. The abstract data shape semantics of a data-processing program P is thus:

$$\Delta^\natural [\![P]\!] = \Delta^\natural \left[\!\left[S^l \right]\!\right] \stackrel{\text{def}}{=} \Delta^\natural [\![S]\!] \left(\lambda p. \begin{cases} \langle \langle \top_{\mathbb{K}_1}, \ldots, \top_{\mathbb{K}_k} \rangle, 1 \cdot [] \rangle & p = l \\ \text{undefined} & \text{otherwise} \end{cases} \right) \quad (5.1)$$

The semantics $\Delta^\natural [\![S]\!] : (\mathcal{L} \to \mathcal{Q}) \to (\mathcal{L} \to \mathcal{Q})$ of each instruction is (equivalently) defined pointwise within $\mathcal{Q}$ in Figure 7: each function $\mathcal{S}^\natural [\![S]\!] : \mathcal{Q} \to \mathcal{Q}$ over-approximates the possible environments and data files that can be read from the program label within the instruction S. The $\text{ASSIGN}_{\mathbb{Q}} [\![X := A]\!]$ operator first invokes $\text{ASSIGN}_{\mathbb{K}_i} [\![X := A]\!]$ on each constraining domain $\mathbb{K}_i$. Then, the $\text{RECORD}(I)$ operation is executed for each input variable $I \in \mathcal{I}$ corresponding to an **input**() sub-expression of A. Finally, the assignment is performed on the input domain by $\text{ASSIGN}_{\mathbb{H}} [\![X := A]\!]$. Similarly, the $\text{FILTER}_{\mathbb{Q}} [\![B]\!]$ operation is first executed on each constraining domain $\mathbb{K}_i$ by $\text{FILTER}_{\mathbb{K}_i} [\![B]\!]$, and then on the input domain by $\text{FILTER}_{\mathbb{H}} [\![B]\!]$. The $\text{REPEAT} [\![A]\!]$, PUSH, POP, $\overline{\text{POP}}$ have no effect on the constraining domains and only modify the input domain (cf. Section 4).

The abstract semantics of each instruction is sound:

Lemma 5. $\mathcal{S} [\![\gamma_{\mathbb{Q}}(Q)]\!] \subseteq \gamma_{\mathbb{Q}}(\mathcal{S}^\natural [\![Q]\!])$

where the concretization function $\gamma_{\mathbb{Q}} \colon \mathcal{Q} \to \mathcal{P}(\mathcal{E} \times \mathcal{D})$ is $\gamma_{\mathbb{Q}}(\langle \langle K_1, \ldots, K_k \rangle, H \rangle) \stackrel{\text{def}}{=} \{ \langle \rho, D \rangle \in \mathcal{E} \times \mathcal{D} \mid \rho \in \gamma_{\mathbb{K}_1}(K_1) \cap \cdots \cap \gamma_{\mathbb{K}_1}(K_k), D \in \gamma_{\mathbb{H}}(H) \}$.

Thus, the abstract data shape semantics $\Delta^\natural [\![P]\!]$ is also sound:

Theorem 1. *For each data-processing program P, we have $\Delta [\![P]\!] \subseteq \gamma_{\mathbb{Q}}(\Delta^\natural [\![P]\!])$.*

6 Implementation

We have implemented our input data shape analysis in the open-source prototype static analyzer Lyra[4]. The implementation is in python and, at the time of writing, accepts data processing programs written in a subset of python without user-defined classes. Programs are expected to be type-annotated, either manually or by a type inference [28].

For the analysis, various constraining domains are available: in addition to the *type*, *sign*, and *string* domains presented in Section 3.1, Lyra is equipped with the *character inclusion* domain [10], as well as the *intervals* [11], *octagons* [35], and *polyhedra* domains [16], which build upon the apron library [32]. A native (non-apron-based) implementation of the intervals domain is also available. For containers (e.g., lists, sets, dictionaries, ...), a summarization-based abstraction [25] is the default. Lists, tuples, and dictionaries can be expanded up to a fixed bound beyond which they are summarized (cf. Section 3.2).

The data shape analysis is performed backwards on the control flow graph of the program with a standard worklist algorithm [37], using widening at loop heads to enforce termination. The precision of the analysis can be improved by running a forward pre-analysis which collects values information about the program variables (e.g., in Figure 1, this would allow the data shape analysis to know the values of the keys of the *grade*2*gpa* dictionary already at line 9 even if the dictionary is not created until line 1, cf. Example 6).

Lyra outputs the analysis results in json format so that other applications (e.g., automated data checking tools [1,38]) can easily interface with it.

Below, we demonstrate the expressiveness of our data shape abstract domain on more examples besides the program shown in Figure 1.

Magic Trick. Let us consider the following python program fragment:

```
1  T = int(input())
2  for x in range(T):
3      l1 = int (input())
4      for i in range(1,l1):
5          input()
6      L1 = list(map(int,input().split()))
7      for i in range(l1+1,5):
8          input()
9      l2 = int (input())
10     for i in range(1,l2):
11             input()
12     L2 = list(map(int,input().split()))
13     for i in range(l2+1,5):
14         input()
```

(from a solution to the *Magic Trick* problem of the Google Code Jam 2014 programming competition[5]). We instantiate our data shape domain $\mathbb{Q}$ with the type constraining domain $\mathbb{K}_{\mathbb{T}}$ and the interval constraining domain $\mathbb{K}_{\mathbb{I}}$. In this

[4] https://github.com/caterinaurban/Lyra

[5] https://codingcompetitions.withgoogle.com/codejam/archive/2014

case, our data shape analysis with $\mathbb{Q}(\mathbb{K}_{\mathbb{T}}, \mathbb{K}_{\mathbb{I}})$, determines that correct data files for the program have the following shape:

			1	2	…
		1	$\langle\text{INT}, [0, \text{inf}]\rangle$		
d_1^1		2	$\langle\text{INT}, [1, 4]\rangle$		
	4	3	$\langle\text{STRING}, [-\text{inf}, \text{inf}]\rangle$	$\langle\text{STRING}, [-\text{inf}, \text{inf}]\rangle$	…
		⋮	…	…	
		6	$\langle\text{STRING}, [-\text{inf}, \text{inf}]\rangle$	$\langle\text{STRING}, [-\text{inf}, \text{inf}]\rangle$	…
		7	$\langle\text{INT}, [1, 4]\rangle$		
	4	8	$\langle\text{STRING}, [-\text{inf}, \text{inf}]\rangle$	$\langle\text{STRING}, [-\text{inf}, \text{inf}]\rangle$	…
		⋮	…	…	
		11	$\langle\text{STRING}, [-\text{inf}, \text{inf}]\rangle$	$\langle\text{STRING}, [-\text{inf}, \text{inf}]\rangle$	…
		⋮	…	…	…

where d_1^1 denotes the first (i.e., and in fact the only) data field 1 of the first data record in the data file. In particular, we know that $1 \leq, d_2^1 \leq 4$ (resp. $1 \leq d_7^1 \leq 4$) from the for loops at lines 4-5 and 7-8 (resp. at lines 10-11 and 13-14). ■

Bird Watching. Let us consider now the following PYTHON program fragment:

```
1 N, M, S = map(int, input().split())
2 pre = [[] for _ in range(N)]
3 for _ in range(M):
4     f, t = map(int, input().split())
5     pre[t].append(f)
6 for n in pre[S]:
7     ...
```

(from a solution to the *Bird Watching* problem of the SWERC 2019-2020 programming competition[6]). We instantiate $\mathbb{Q}$ with the type constraining domain $\mathbb{K}_{\mathbb{T}}$ and the octagon constraining domain $\mathbb{K}_{\mathbb{O}}$. A forward numerical pre-analysis with the octagon domain $\mathbb{O}$ determines, in particular, that $0 \leq \mathsf{len}(pre) \leq N - 1$ (cf. line 2). Thus, our backward data shape analysis with $\mathbb{Q}(\mathbb{K}_{\mathbb{T}}, \mathbb{K}_{\mathbb{O}})$ determines that correct data files for the program have the following shape:

		1	2	3
	1	$\langle\text{INT}, 0 \leq d_1^1\rangle$	$\langle\text{INT}, 0 \leq d_1^2\rangle$	$\langle\text{INT}, 0 \leq d_1^3 \leq d_1^1 - 1\rangle$
d_1^2	2	$\langle\text{INT}, \text{true}\rangle$	$\langle\text{INT}, 0 \leq d_2^2 \leq d_1^1\rangle$	
	3	$\langle\text{INT}, \text{true}\rangle$	$\langle\text{INT}, 0 \leq d_3^2 \leq d_1^1\rangle$	
	⋮	…	…	

where d_i^j denotes the data field j of the data record i. In particular, we know that $0 \leq d_1^3 \leq d_1^1 - 1$ from the list access at line 6 and, similarly, $0 \leq d_i^2 \leq d_1^1$, for $2 \leq i$, from the list access at line 5. ■

[6] https://swerc.eu/2019/

Adult Census Data. Let us consider the following fragment of a pre-processing PYTHON function for the Adult Census dataset[7]:

```
1  def pre_process_data (data):
2      new_data = []
3      for i in range(len(list(data))):
4          person_new = []
5
6          person_new.append(data[i][0])
7
8          w = data[i][1]
9          if w == "Private":
10            person_new.append(w)
11         elif w == "Self-emp-not-inc" or w == "Self-emp-inc":
12              person_new.append("Self-Employed")
13         elif w == "Federal-gov" or w == "Local-gov" or w == "State-gov":
14              person_new.append("Government")
15         elif w == "Without-pay" or w == "Never-worked":
16              person_new.append("Other")
17         else: raise Exception("Workclass not matched: ", w, i)
18
19         ...
20
21         new_data.append(person_new)
22
23     return new_data
```

(taken from [39]) where the function argument data has been loaded from a CSV file. Our backward shape analysis instantiated with the string set constraining domain $\mathbb{K}_{\mathbb{W}}$ determines that correct CSV files have the following shape:

	1	2	...
1	$\top_{\mathbb{W}}$	W	...
2	$\top_{\mathbb{W}}$	W	...
3	$\top_{\mathbb{W}}$	W	...
⋮	...	...	...

where W is the set of strings { 'Private', 'Self-emp-not-inc', 'Self-emp-inc', 'Federal-gov', 'Local-gov', 'State-gov', 'Without-pay', 'Never-worked' }. ■

7 Related Work

Learning the input format of a given program is not a new research area but it has recently seen increased interest, especially in the contest of grammar-based automated test generation and fuzzing applications [23,30,34, etc.].

Many of the approaches in the literature are *black-box*, e.g., GLADE [3] and LEARN&FUZZ [24]. These generally generate input grammars or grammar-like structures that are strictly meant as intermediate representation to be fed to a test generation engine and are not meant to be readable by a human. On the other hand, the result of our analysis is human-readable and can be used for other purposes than test generation, e.g., code specification and data cleaning. Moreover, these approaches have to rely on samples of valid inputs, while our approach only needs the program to be analyzed.

[7] https://archive.ics.uci.edu/ml/datasets/adult

Another sample-free approach is AUTOGRAM [31], which uses dynamic data flow analysis to generate readable and usable grammars. One disadvantage of this approach is that it will skip parts of the input if these are not stored in some program variables (e.g. if a program scans over a comment). On the contrary, in such a case, our approach will not know any value information about the skipped data but will still know that this data should be present in the data file (see the *Magic Trick* example in Section 6 for instance).

To the best of our knowledge ours is the first approach that uses static analysis to infer the input format of a given program. Moreover, contrary to the above grammar synthesis approaches, our approach infers *semantic* (and not just syntactic) information on the input data of a program. Closest to ours, is the work of Cheng and Rival [8] on the static analysis of spreadsheet applications. They however only focused on type-related properties.

Finally, the main difference compared to the inference of necessary preconditions proposed by Cousot et al. [14] or the (bi-)abduction [6] used in tools like INFER [5] is that our analysis can also deal with inputs read at any point during the program (thus notably also inside loops whose execution may depend on other inputs — this is where the need for the stack comes from, cf. Section 4).

8 Conclusion and Future Work

In this paper, we have proposed a parametric static shape analysis framework based on abstract interpretation for inferring semantics properties of input data of data-processing programs. Specifically, our analysis automatically infers necessary conditions on the structure and values of the input data for the data-processing program to run successfully and correctly.

It remains for future work to explore possible applications of the result our analysis. In particular, we are interested in developing better grammar-based testing approaches. We are also interested in developing tools for assisting and guiding or even automating the checking and cleaning of data.

References

1. R. S. Abdelbar. *Automated Checking of Implicit Assumptions on Textual Data*. Bachelor's thesis, ETH Zurich, Switzerland, 2018.
2. V. Arceri and I. Mastroeni. An Automata-based Abstract Semantics for String Manipulation Languages. In *VPT@Programming*, pages 19–33, 2019.
3. O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. In *PLDI*, pages 95–110, 2017.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In *The Essence of Computation*, pages 85–108, 2002.
5. C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving Fast with Software Verification. In *NFM*, pages 3–11, 2015.

6. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of the ACM*, 58(6):26:1–26:66, 2011.
7. D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of Pointers and Structures. In *PLDI*, pages 296–310, 1990.
8. T. Cheng and X. Rival. Static Analysis of Spreadsheet Applications for Type-Unsafe Operations Detection. In *ESOP*, pages 26–52, 2015.
9. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for High-Level Web Service Construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.
10. 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.
11. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Programs. In *Symposium on Programming*, pages 106–130, 1976.
12. 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.
13. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2&3):103–179, 1992.
14. P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic Inference of Necessary Preconditions. In *VMCAI*, pages 128–148, 2013.
15. 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.
16. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96, 1978.
17. A. Cox, B. E. Chang, and X. Rival. Automatic Analysis of Open Objects in Dynamic Language Programs. In *SAS*, pages 134–150, 2014.
18. A. Cox, B. E. Chang, and S. Sankaranarayanan. QUIC Graphs: Relational Invariant Generation for Containers. In *ECOOP*, pages 401–425, 2013.
19. D. Delmas and A. Miné. Analysis of Software Patches Using Numerical Abstract Interpretation. In *SAS*, pages 225–246, 2019.
20. I. Dillig, T. Dillig, and A. Aiken. Fluid Updates: Beyond Strong vs. Weak Updates. In *ESOP*, pages 246–266, 2010.
21. I. Dillig, T. Dillig, and A. Aiken. Precise Reasoning for Programs Using Containers. In *POPL*, pages 187–200, 2011.
22. J. Fulara. Generic Abstraction of Dictionaries and Arrays. *Electronic Notes in Theoretical Computer Science*, 287:53–64, 2012.
23. P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-Based Whitebox Fuzzing. In *PLDI*, pages 206–215, 2008.
24. P. Godefroid, H. Peleg, and R. Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. In *ASE*, pages 50–59, 2017.
25. D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and S. Sagiv. Numeric Domains with Summarized Dimensions. In *TACAS*, pages 512–529, 2004.
26. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
27. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.
28. M. Hassan, C. Urban, M. Eilers, and P. Müller. MaxSMT-Based Type Inference for Python 3. In *CAV*, pages 12–19, 2018.
29. M. Hennessy and J. F. Power. An Analysis of Rule Coverage as a Criterion in Generating Minimal Test Suites for Grammar-Based Software. In *ASE*, pages 104–113, 2005.

30. C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. In *USENIX Security*, pages 445–458, 2012.
31. M. Höschele and A. Zeller. Mining Input Grammars from Dynamic Taints. In *ASE*, pages 720–725, 2016.
32. B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, page 661—667, 2009.
33. J. Liu and X. Rival. An Array Content Static Analysis Based on Non-Contiguous Partitions. *Computer Languages, Systems & Structures*, 47:104–129, 2017.
34. R. Majumdar and R. Xu. Directed Test Generation using Symbolic Grammars. In *ASE*, pages 134–143, 2007.
35. A. Miné. The Octagon Abstract Domain. *Higher Order and Symbolic Computation*, 19(1):31–100, 2006.
36. R. Monat, A. Ouadjaout, and A. Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In *ECOOP*, 2020.
37. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
38. M. Schumacher. *Automated Generation of Data Quality Checks*. Master's thesis, ETH Zurich, Switzerland, 2017.
39. C. Urban, M. Christakis, V. Wüstholz, and F. Zhang. Perfectly Parallel Fairness Certification of Neural Networks. *CoRR*, abs/1912.02499, 2019.