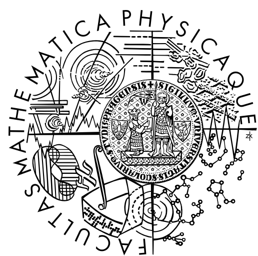

FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

Jan Hrubý

Abstract Interpretation of Pandas

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Tomáš Petříček, Ph.D.
Study programme: Computer Science

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

(Not generated by ChatGPT even though their suggestions were nicer) There are people I would not be able to finish the thesis without and I think they deserve to be mentioned. The first person is my supervisor, Professor Tomáš Petříček. I thank him for his guidance, great insight and courage he provided me with. I would also like to thank my fiancée Lana. She was there for me whenever I needed her, and I am grateful for that. I would also want to thank the tearoom Jedna Báseň and tearoom Dharmasala for a comfortable spot for writing my thesis and a steady supply of great tea.

Title: Abstract Interpretation of Pandas

Author: Jan Hrubý

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D., Department of Distributed and Dependable Systems

Abstract: Pandas is a Python library widely used for data-manipulation. The code written with Pandas lacks any type-safety and everything is decided at runtime. This can potentially be a source of errors and crashes at runtime. One way to deal with that is to use another, type-safe, language and a library with better safety guarantees and compile-time checks. This approach is not widely used as it is not very user-friendly. An alternative approach could be to use program verification method Abstract Interpretation to perform some checks before the run of the program. The goal of this thesis is to design a framework for analyzing data-manipulation programs and implement an analyzer for the Pandas library. The framework will be based on the Abstract Interpretation. The capabilities of the resulting analyzer will be evaluated on a set of small but realistic case studies.

Keywords: python, pandas, abstract interpretation, data analysis

Název práce: Abstraktní interpretace knihovny Pandas

Autor: Jan Hrubý

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Pandas je oblíbená knihovna pro manipulaci a analýzu dat v Pythonu. Kód napsaný s Pandas nemá žádné typové kontroly a vše je rozhodováno za běhu programu. To může být zdrojem chyb a pádu celého programu za běhu. Jeden ze způsobů jak tento problém řešit je použít jiný, staticky typovaný, jazyk a knihovnu se silnějšími bezpečnostními zárukami a kompilačními kontrolami. Takové řešení se však nerozšířilo kvůli horší uživatelské přívětivosti. Alternativní cesta by mohla být použítí metody Abstraktní Interpretace ke kontrole programu před jeho během. Cílem této práce je navrhnout způsob jak využít Abstraktni Interpretaci k analýze programů pro manipulaci s daty, a implementovat analyzátor pro knihovnu Pandas. Schopnosti implementovaného analyzátoru budou zhodnoceny na několika malých avšak realistických případových studiích.

Klíčová slova: python, pandas, abstraktní interpretace, datová analýza

Contents

Introduction **7**

1 Abstract Interpretation **9**
1.1 Introductory Example . . . 9
1.2 Lattices . . . 10
1.3 Program semantics . . . 12
1.4 Galois Connections . . . 13
1.5 The method . . . 14
Summary . . . 16

2 Data manipulation and Pandas library **17**
2.1 Data structures . . . 17
2.1.1 Series . . . 17
2.1.2 Dataframe . . . 19
2.2 Common operations . . . 20
2.2.1 Relational operations . . . 20
2.2.2 Vectorized operations . . . 23
Summary . . . 24

3 Putting it all together **25**
3.1 The concrete lattice . . . 25
3.2 The abstract lattice . . . 26
3.3 The Galois Connection . . . 26
3.4 Abstract Operations . . . 27
3.5 Adding other types . . . 28
3.6 Final proposal . . . 28
3.7 Limitations . . . 30
Summary . . . 30

4 Pandalyzer **32**
4.1 The Goal . . . 32
4.2 Architecture . . . 33
4.2.1 The high-level idea . . . 33
4.2.2 Design Decision - Parsing and AST . . . 33
4.2.3 Analysis Context . . . 34
4.2.4 Python structures representation . . . 35
4.2.5 Nondeterminism, unknown values and error recovery . . . 35
4.2.6 Dataframe and Series Representation . . . 36
4.2.7 The Abstract Operations . . . 37
4.2.8 Design Decision - Configuration file . . . 39
Summary . . . 40

5 Evaluation of the solution **42**
5.1 Case Studies . . . 42
Summary . . . 52

Conclusion **53**
Related Work . . . 53
Future Work . . . 54

Bibliography **55**

List of Figures **57**

List of Abbreviations **59**

A User documentation **60**
A.1 Building from source . . . 60
A.2 Running the tool . . . 60

Introduction

An important part of data analysis is called data manipulation or data wrangling. The goal of data manipulation is to take the raw data obtained from some source, clean them and transform them into better-structured error-free data which are more suitable for the data analysis itself. The data manipulations - cleaning and transformations - usually cannot be done manually, since the data are too large for that. For manipulating large amounts of data, one needs an automated tool. Popular approach to solve this problem is a library in a programming language. Data manipulation libraries that became widely used in the industry, are Pandas in Python[1], Tibble in R[2] (R also has built-in support for data frames[3]) or DataFrames.jl in Julia[4]. These libraries usually come with two types of data structures: one-dimensional array usually called Series and two-dimensional tabular structure usually called Dataset or Dataframe.

The mentioned languages, Python, R and Julia, are all dynamic and interpreted languages, meaning that they do not provide the programmer with any compile-time checks for type consistency and variable or column existence. These languages are popular in data-related subjects due to their easy syntax and the fact that one does not need to complete a course on computer architecture to use it. However, the user of a dynamic languages has to keep most of the information in their head (or use comments in code a lot) which is a potential source of many different errors, that are, in addition, detected no sooner than at runtime when the program crashes. Consider for example the program in Listing 1. There are some harder-to-spot mistakes such as referencing a dropped column, summing columns of different types or a misspelled column name. All these mistakes are detected at runtime causing crash of the program.

```
import pandas as pd

df = pd.read_csv("data.csv")
df_copy = df
df_copy.drop("column1", inplace=True)

grouped = df.groupby("column1")
# Error - column1 does not exist already

final_score = df["score_a"] + df["score_b_note"]
# Error - summing Series of ints with strings

print(df["colunm2"])
# Error - misspelled column name colunm2
```

Listing 1 Pandas code with errors

A potential approach to prevent these issues could be to use a different programming language. One with a static strict type system, that would allow us to use typed data-frames - a data frames with statically assigned column names and types. The results of data-frame transformations would then be determined by the function type signature. Even though this solution would solve the issue with column naming and column types, it has many problems that dynamic languages do not have. The arguments mentioned for the popularity of dynamic languages are also arguments against static languages. Namely, a sharp learn-

ing curve, a complicated syntax, and a technical complexity. Additionally, the necessity for a lot of boilerplate code, worse readability and the fact, that dynamic languages, contrary to the static languages, usually support a REPL tool or just creating one file and running it. Ideally, we would like to use dynamic languages like Python, R or Julia, while keeping the safety of static languages.

Thesis goals

The goal of this thesis is to propose and explore an alternative solution to the problem of data-manipulation code being error-prone and provide an implementation of such solution. The solution will be based on a program verification method called Abstract Interpretation. The origins of the ideas behind the method can be traced back to the 1970s when it was developed by Patrick and Radhia Cousot[5].

In essence, Abstract Interpretation partially executes the given computer program over an abstract domain (such as {+, 0, -} for integers). It gains partial information about the programs semantics and answers questions (as mentioned by B. Blanchet[6]) such as:

- What is the worst-case execution time?
- Is the program secure against a specific attack?
- How much stack will the program use?
- Which parts of the code are unreachable?

The idea of analyzing programs working with dataframes was already proposed by Yungyu Zhuang and Ming-Yang Lu [7]. However, we aim to present more fundamental analysis and develop a more comprehensive tool. We will define our own framework with more levels of abstraction and expand the abstraction to non-pandas types as well. Also, we will interpret both branches of if-statements in case that we do not have enough information to choose one, and we plan to be able to work with more uncertainty from the user.

In the first chapter of the thesis, we will review the theory of POSETs, Lattices and Galois connection which are the core mathematical concepts of Abstract Interpretation. Then, in the second chapter, we will explore the needs of data scientists regarding the tools for data manipulation. We will also take a look at how Pandas provide the data scientists with these needs. The third chapter will put all the pieces together. We will define an Abstract Interpretation framework for the world of Dataframes and Series. We will also take a look at how some basic operations like merge, groupby or concatenation will work in the abstract domain. The fourth chapter will be dedicated to the implementation of the analysis tool itself, its architecture, documentation, capabilities, and limitations. Finally, in the last, fifth, chapter, we will evaluate the implemented solution on a set of realistic examples of Pandas code and analyze and discuss its usability in practice.

1 Abstract Interpretation

In this chapter we explain the Abstract Interpretation from the theoretical point of view. At first, we recall some basic mathematical definitions such as partial order or supremum and infimum. Then we use this knowledge to define lattices and Galois connections, that are the core concepts of Abstract Interpretation. We also formalize a program—its syntax, semantics, trace and Collecting semantics. At last, we use all this knowledge to explain the Abstract Interpretation method. This chapter is greatly inspired by Introduction to Abstract Interpretation by B. Blanchet [6]. He also covers more advanced topics that are not needed for our purposes, so I would refer the more curious readers to his article.

1.1 Introductory Example

We start with a very simple example that shows the basic idea behind abstract interpretation. I do not define the syntax of the programming language for now as I assume that it is mostly clear. Consider this very simple toy program:

```
a = read_number()
b = read_number()
c = a + b
d = a * b
e = a * a
f = a - b
g = a / b
```

Given the assumption that `read_number()` returns one real number given by the user and does not fail, what can we say about the content of the variables `c`, `d`, `e`, `f` and `g`? The answer is not much, except the fact that `e` is non-negative (it is a square) and the calculation of `g` can fail (due to division by zero). We cannot even say whether the values will be greater or less than the inputs `a` and `b`.

This, of course, changes, when we introduce additional constraints on the inputs `a` and `b`. Example of such constraints can be: `Both a and b are positive`. What can we say about the variables then?

- `c` is positive and greater than `a` and `b`
- `d` is positive
- `e` is positive
- `f` is in the interval $(-b, a)$
- `g` is positive and the computation will not fail

Notice that the approximation of `d` and `e` would be even more precise if we also said whether the inputs are greater or less than 1.

This example shows that when we know the approximations of the input values, we are able to make statements about the behavior of the program. In abstract interpretation, we try to interpret (execute) the given program with approximated input values and try to prove that the output is in certain bounds or that it does not crash in any (or some) case. Such analysis is always correct, but the results are not precise—it is an approximation of the behavior of the program. Also note that this can be actually useful in practice since in many programs, the variables usually are bounded due to their semantics. For example, the hour of a day will always be a number between 0 and 24 (The answer to the question of why not just 23 is left as an exercise to the reader), number of workers in a factory will be always non-negative, election results of a political party will always be between 0 and 100 and a sum of degrees of an undirected graph will always be an even number.

But what information should we remember for the variables in the program? In the simplest case, we can just remember whether the values are positive, negative or zero. More advanced case (the one that we use in the example above) can use intervals. Generally, we use a union of arbitrary disjunctive (for simplicity) sets. Then the input constraints can look like this:

$$a \in 1, 2, 3 \cup (0, 1)$$
$$b \in \mathbb{R}$$

The idea of approximation of values with unification of sets of possible solutions can be applied to other subjects than just real numbers. Some examples:

- Matrix operations can be approximated by dimensions of matrices and ensure that the operations (matrix addition, multiplication, gaussian elimination) are correct.

- Geometry in 2D (or higher dimensions). Points in the context of 2D (or higher dimensional) geography can be approximated by axis-aligned rectangular sectors.

Another possible application—Tabular data—will be covered in the chapter 3.

1.2 Lattices

We first recall few already well-known definitions (Poset, Supremum, Infimum) and then use this to state the definitions of more advanced objects (Lattice, Complete Lattice, Galois Connection) that will be later essential for formally defining the Abstract Interpretation framework.

Definition 1 (Partial order)**.** *Relation $\leq$ on a set S is a partial order if $\forall a, b, c \in S$:*

1. *$a \leq a$ (Reflexivity)*

2. *$a \leq b \wedge b \leq a \implies a = b$ (Antisymmetry)*

3. *$a \leq b \wedge b \leq c \implies a \leq c$ (Transitivity)*

Then defining a partially ordered set (Poset) is straightforward:

Definition 2 (Poset)**.** *The pair* $(S, \leq)$ *is a Poset, if* $\leq$ *is a partial order on* S

Example. We also mention few examples of Poset:

- $(\mathbb{R}, \leq), (\mathbb{Q}, \leq), (\mathbb{Z}, \leq)$ are all Posets
- For a set S, $(\mathbb{P}(S), \subseteq)$ is a Poset
- Non-negative integers with divisibility are a Poset
- For directed acyclic graph $G = (V, E)$ the pair $(V, \mathit{reachability})$ is a Poset

Last thing that we need before defining a Lattice are the definitions of supremum and infimum.

Definition 3. *On partially ordered set* $(S, \leq)$*, for* $R \subseteq S$*:*
Upper bound of R *in* S *is* $a \in S$ *such that* $\forall x \in R : x \leq a$*.*
Lower bound of R *in* S *is* $a \in S$ *such that* $\forall x \in R : a \leq x$*.*
An upper bound s *of* R *in* S *is called a supremum (we use the symbol* sup*) if for all upper bounds* b *of* R *in* S *holds that* $s \leq b$*.*
A lower bound w *of* R *in* S *is called an infimum (we use the symbol* inf*) if for all lower bounds* b *of* R *in* S *holds that* $w \geq b$*.*

With this knowledge, we have all we need to define a Lattice:

Definition 4 (Lattice)**.** *A Poset* $(L, \leq)$*) is a Lattice if:*

$$\forall a, b \in L\ \exists s \in L : s = \sup(\{a, b\})$$

$$\forall a, b \in L\ \exists u \in L : u = \inf(\{a, b\})$$

In other words, Lattice is a poset, where each two elements of a Lattice have a supremum and an infimum.

Definition 5 (Complete Lattice)**.** *A Poset* $(L, \leq)$ *is a Complete Lattice if:*

$$\forall X \subseteq L\ \exists s \in L : s = \sup(X)$$

$$\forall X \subseteq L\ \exists u \in L : u = \inf(X)$$

This means that each subset of a Complete Lattice has a supremum and an infimum. Note that this is a stronger condition than at Lattices, since pair is also a subset. That also means that every Complete Lattice is a Lattice.

Example. All $(\mathbb{R}, \leq), (\mathbb{Q}, \leq), (\mathbb{Z}, \leq)$ are Complete Lattices.
Also, a power set of a set with inclusion as an ordering is a Complete Lattice. The supremum of a set of sets on a power set can be defined as the union of the sets, and the infimum can be defined as their intersection.
The pair $(V, \mathit{reachability})$ for a directed acyclic graph is **not** a lattice. For example, if we take vertices from different weakly connected components, they will not have a supremum and infimum.

1.3 Program semantics

To properly talk about analyzing programs, we first need to define what a program is. We define a simple syntax, and then we add its semantics. So let us start by stating a few definitions regarding the syntax:

Definition 6 (Expression)**.** *Expression has the following recursive definition:*

- $\forall n \in \mathbb{Z}$, *n is an expression*
- *a variable x is an expression*
- $\forall E, F$ *expressions, the following are expressions:* $E + F$, $E - F$, $E \geq F$, $E \cdot F$

Definition 7 (Statement)**.** *The following are Statements:*

- *halt*
- $x := E$ *for a variable x and an expression E*
- $if\ E\ goto\, n$ *for an expression E and $n \in \mathbb{Z}$*
- *input x for a variable x*
- *print E for an expression E*

In our model, we represent programs as a function that maps program locations (think addresses in memory) to statements (CPU instructions) at the given location.

Definition 8 (Program)**.** *Program is a function $Prog$ from integers to Statements. Formally:*

$$Prog : \mathbb{Z} \rightarrow Statements$$

The state of the running program (process) at a given time t is given by the program counter (*pc*), saying which command is to be executed next, and the set of the variables in the environment Env. The environment Env is a table that assigns values to variables in a scope.

To know how the *pc* and the variables change during the run of the program, we must define the semantics of the expressions and statements.

Definition 9 (Expression semantics)**.** *The semantics (result) of an Expression E is $[E] =$:*

- n, *if* $E = n$
- $Env[x]$, *if* $E = x$
- $[F] + [G]$, *if* $E = F + G$ *(similarly for $F \cdot G$, $F - G$)*
- 1 *if* $E = F \geq G$ *and* $[F] \geq [G]$
- 0 *if* $E = F \geq G$ *and* $[F] < [G]$

Definition 10. *Statement semantics*

- *halt - ends the execution of the process*
- $x := E$ *- sets the value of a variable x in the environment to the value* $[E]$
- $if\ E\ goto\ n$ *- sets the pc to n if the result of E is non-zero*
- $input\ x$ *- sets value of variable x in the environment to the user input*
- $print\ E$ *- adds result of E to the user output*

For a given program and user input, the process goes over a sequence of states. We call such sequence of states a *trace*. Since the trace depends on the user input, the semantics of the program is given by the set of traces, rather than a single trace. We define **Collecting Semantics** as an extension of the standard semantics on the set of all user possible inputs. So Collecting Semantics is the semantics capable of computing the set of all possible traces.

1.4 Galois Connections

Let us this time start by the definition:

Definition 11 (Galois Connection)**.** *Let L_1 (with ordering $\leq_1$) and L_2 (with ordering $\leq_2$) be Lattices. Also, let $\alpha : L_1 \to L_2$ and $\gamma : L_2 \to L_1$. Then the tuple (α, γ) is a Galois Connection, if:*

$$\forall x \in L_1,\ \forall y \in L_2 :\ \alpha(x) \leq_2 y \leftrightarrow x \leq_1 \gamma(y)$$

Galois Connection is a pair of functions connecting two Lattices. We call the first Lattice (L_1) the **concrete lattice** and the second lattice (L_2) the **abstract lattice**. Then the function α is called the **abstraction** and γ is the **concretization**.

In the context of Abstract Interpretation, the concrete lattice is a power set (ordered by inclusion) of a set of values of variables of the program, while the abstract lattice is the set of values we work with during the program analysis.

Examples of Galois Connections

1. Approximating real numbers with signs and zero:

   The concrete lattice is $\mathbb{P}(\mathbb{R})$ and the abstract lattice is $\mathbb{P}(\{+, -, 0\})$ (ordered by inclusion). The elements of abstract lattice say what signs the corresponding element of concrete lattice can have. For example, the set $\{+, 0\}$ represents a non-negative number and $\{+, -\}$ represents a non-zero number in the concrete lattice.

   The definition of abstraction function is as follows:

$$\alpha(C) =$$
$$\{+\ if\ C\ contains\ positive\ number\} \cup$$
$$\{-\ if\ C\ contains\ positive\ number\} \cup$$
$$\{0\ if\ C\ contains\ zero\}$$

And the definition of concretization function is defined:

$$\begin{gathered}\gamma(A) = \\ (-\infty, 0)\, if - \in A \cup \\ 0\, if\, 0 \in A \cup \\ (0, \infty)\, if + \in A\end{gathered}$$

2. Abstracting geometric points in 2D to axis-aligned rectangular areas

   The concrete lattice is $\mathbb{P}((x, y) \in \mathbb{R}^2)$ ordered by inclusion. The abstract lattice is the set of all axis-aligned rectangles given by the two points (x_1, y_1) and (x_2, y_2), where $x_1 \leq x_2$ and $|y_1 \leq y_2$. We denote such a rectangle $Rect((x_1, y_1), (x_2, y_2))$.

   Then the definition of the abstraction is as follows:

$$\begin{gathered}\alpha(B) = Rect(\\ (min\ x\ fromB, min\ y\ fromB), \\ (max\ x\ fromB, max\ y\ fromB) \\ )\end{gathered}$$

   And the definition of the concretization function is:

$$\begin{gathered}\gamma(Rect((x_1, y_1), (x_2, y_2))) = \\ \{(x, y) \forall x \in [x_1, x_2], \forall y \in [y_1, y_2]\}\end{gathered}$$

3. Reducing matrices to their dimensions:

   The concrete lattice is the power set of a set of all real matrices. The abstract lattice is a power set of a set of pairs of integers (r,s) representing the matrix dimensions (number of rows and columns).

   The definition of the abstraction is:

$$\alpha(M) = \{(r, s) \forall X \in M, X \in \mathbb{R}^{r \times s}\}$$

   The definition of the concretization is:

$$\gamma(M) = \{X \forall X \in \mathbb{R}^{r \times s}, (r, s) \in M\}$$

Note that the set of all traces ordered by inclusion form a complete lattice.

1.5 The method

In the previous parts of this chapter, we defined Lattices. We defined the concrete lattice—the normal world where computer runs. We also defined the abstract lattice—the world where the analysis takes place. Then we created the semantics of the statements and the expressions in the concrete lattice. Finally, we

connected the two worlds—concrete and abstract lattice—with Galois Connection, which allows us to traverse between the two.

What we are still missing is the semantics of the statements and expressions in the abstract lattice. This is where we finally use the fact that the connection between the concrete and abstract lattice is a Galois Connection. We show that the abstract semantics can be derived from the semantics of the concrete lattice and the Galois Connection, which is the last piece of the Abstract Interpretation puzzle. We first define the abstract semantics of values, operators, and environment variables. Then we move to the expressions and statements.

- **Values**

  This is the simplest case, since the abstract semantics of n from the concrete lattice is just $\alpha(n)$.

- **Operators**

  The abstract semantics of the operator *op* will be (for a and b from abstract lattice):

  $$a\ op\ b = \alpha(\gamma(a)\ op\ \gamma(b))$$

  So we concretize both the elements from the abstract lattice, then apply the operation *op* and abstract the result back. It can be shown that this definition is the best (meaning smallest) possible approximation of the concrete *op*. For more details, see the paper Introduction to Abstract Interpretation by B. Blanchet[6]

- **Environment Variables**

  The variable $x \in Env$ can be abstracted to x_a by just taking the abstraction of x ($x_a = \alpha(x)$).

- **Compound Expressions**

  The abstract semantics of the compound expressions (sums, products...) can be obtained just by replacing the concrete semantics of the operator with the associated abstract operator.

- **Statements**

  - If $Prog(pc) = x := E$ then $pc = pc + 1$, $Env_a[x] = [E]_a$.
  - If $Prog(pc) = input\ x$ then $pc = pc + 1$, $Env_a[x] = \mathbb{R}$.
  - The case, when $Prog(pc) = if\ E\ goto\ n$ is a bit tricky. There are two possibilities of what can happen. The first is that the $[E]_a$ contains only zero, or it does not contain zero at all. In that case, we know which branch should be taken, and we just update the *pc* accordingly. The other case happens when the $[E]_a$ contains both zero and non-zero elements. Then we do not know which branch should be taken, so the execution must split into two branches, and both options must continue separately.
  - If $Prog(pc) = halt$ then we just stop the execution of the program (or the given branch if the execution was split).

– If $Prog(pc) = print\,x$ then we print the set $Env_a[x]$.

So how does the method work? We abstract the user inputs and then interpret the program over the abstract lattice (hence the name Abstract Interpretation) using the abstract definitions of Expressions, Statements, etc. We branch when we do not know what the next state should be (if statements). Then, when we want to know some property of the program, we just observe the abstract version of the variables we care about or check whether some branches halted with failure.

Summary

Abstract Interpretation is a program analysis method. We can imagine it as approximating the interpretation of the program (like normal interpreters do) with incomplete information about the user input, branching when we do not know. Instead of a standard semantics of a program, we take a collecting semantics as an extension over all possible user inputs, and we abstract the collecting semantics via an abstraction function of Galois Connection. Then we interpret the program considering the abstract semantics.

2 Data manipulation and Pandas library

In this chapter, our goal is to explore the world of data manipulation. We explain what data structures are usually used and what the common operations on them look like. We will need the information in the next chapters when defining the Abstract Interpretation framework for these data structures. We show the concepts on Pandas and discuss what Pandas does differently.

2.1 Data structures

When we talk about data structure, we usually define it as a way of organizing data in the computer memory. However, there are two concepts to distinguish—the interface and the implementation.

The interface is a set of operations that we are allowed to do with the data structure as users. A Good example of a data structure interface can be an Array, List, Dictionary or a Heap. Implementation on the other hand is how the data structure works under the hood to provide the interface promised. To give an example of an implementation of a data structure, I mention a binary tree, n-ary heap or a linked list.

In this chapter, when talking about data structures, we omit the implementation details, and we focus only on the interface of the data structure, i.e., what operations are provided. Also, we assume the existence of primitive data types such as integers, floating-point numbers, strings etc.

Definition 12 (List interface)**.** ***List*** *interface is a set of methods for random access, appending, inserting, updating and removing elements from a collection. All operations use numeric zero-based indexing.*

Definition 13 (Dictionary interface)**.** ***Dictionary*** *interface supports accessing, adding, replacing or removing elements from a collection. All operations use key-based indexing.*

2.1.1 Series

The first and the simplest data structure is usually called Series. In its simplest form, it is a one-dimensional data structure that holds data of some primitive data type. It supports a List interface, so random access based on integer indexing is supported as well as adding, removing and modifying items.

The Listing 2.1 shows basic work with the Series data structure in Pandas.

```
import pandas as pd
# Creating Series
series = pd.Series([1, 1, 2, 3, 5, 8])

# Adding an element
series[6] = 13

# Removing an element at index 0
```

```python
series.drop(0, inplace=True)

# Replacing an element
series[1] = 100
```

Listing 2.1 Working with Series in Pandas

There can also be an index associated with the Series. The index is a set of distinct values of any primitive type (usually a string or a time) associated with the values of the Series. It expands the interface of a Series with a Dictionary interface. Consequently, the items can be accessed using the values in the index. Another usual feature of a Series is an optional label describing the data. Listing 2.2 shows how the work with index works in Pandas.

```python
import pandas as pd

# Creating Series with index
data = [1, 2, 3, 4]
index = ["first", "second", "third", "fourth"]
series = pd.Series(data=data, index=index)

# Accessing based on index
print(series["second"]) # Prints 2

# Adding based on index
series["fifth"] = 5

# Removing based on index
series.drop("fifth", inplace=True)
```

Listing 2.2 Index on a Series

Contrary to what we said, Series in Pandas is a heterogeneous data structure—it is able to hold values of different types together. So the code showed in Listing 2.3 is completely valid Pandas code.

```python
import pandas as pd

# Create a Series of values with different types
series = pd.Series([0, True, "two", "three", 4])
```

Listing 2.3 Heterogenous Series

The explanatory visualization of a Series can be found in the Figure 2.1

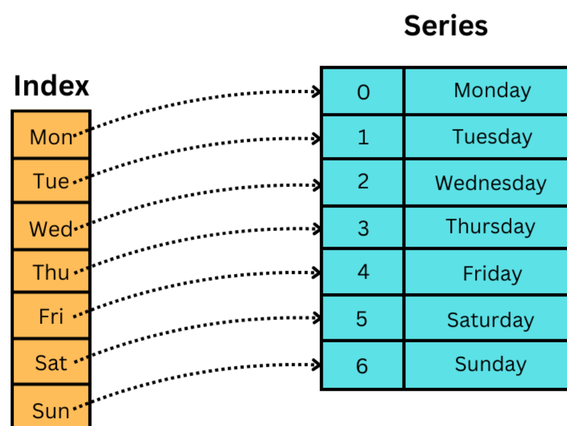

Figure 2.1 Schema of a Series

2.1.2 Dataframe

Dataframe is a two-dimensional tabular data structure. A good way to look at a Dataframe is to see it as a Dictionary, where the keys are names of the columns of a table and values are Series representing the columns themselves. This also means that the Dataframe supports indexing of columns based on column names and integer-based indexing of rows. Listing 2.4 shows basic work with the Dataframe in Pandas.

```
import pandas as pd

# Create a Dataframe from a map of columns
data = {"column1": [1, 2, 3], "column2": ['a', 'b', 'c']}
dataframe = pd.DataFrame(data)

# Add a new row
dataframe.loc[3] = [4, 'd']

# Add a new column
dataframe['column3'] = [True, False, True, False]

# Replace a value
dataframe.at[1, 'column3'] = True
```

Listing 2.4 Working with Dataframe in Pandas

The Dataframe can also have an index associated with the rows of Series. Consequently, the Dataframe supports Dictionary interface on both rows and columns. Adding of new columns and rows is also supported. The listing 2.5 shows how we can use indexes when working with Dataframes in Pandas.

```
import pandas as pd

# Create a dataframe with index
dataframe = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
}, index=['x', 'y', 'z'])

# Access elements via index
print(dataframe.at['x', 'A']) # Prints 1

# Change value using index
dataframe.at['z', 'C'] = -9
```

Listing 2.5 Index on a Dataframe

The explanatory visualization of a Dataframe can be found in the Figure 2.2

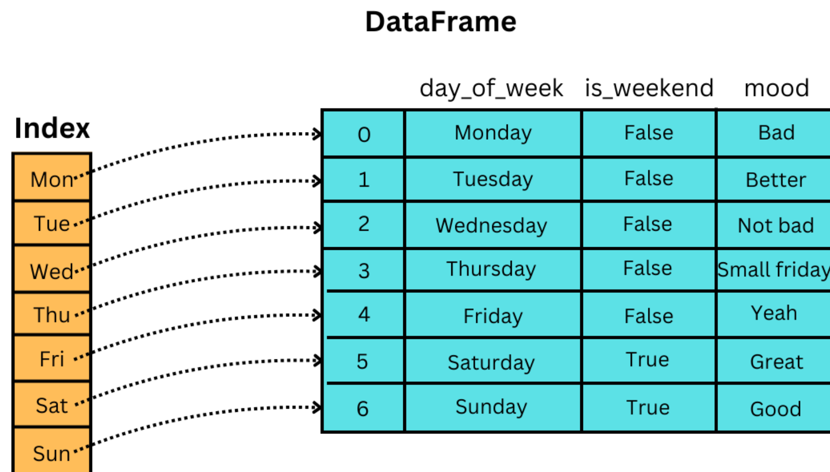

Figure 2.2 Schema of a Dataframe

The Dataframe is usually loaded from a CSV file via the `read_csv` function, and the Dataframe that is the result of the computation is usually stored to another CSV file via the `to_csv` function.

2.2 Common operations

To actually manipulate the data into some useful form, we need operations that are more powerful than just indexing and adding and removing elements. The operations that are commonly used in data manipulation were greatly influenced by the operations on relational databases and the SQL language. However, they are usually more flexible and customizable.

2.2.1 Relational operations

All well-known SQL relational databases support the following operations: `SELECT`, `GROUP BY`, `HAVING`, `WHERE`, `JOIN`, `ORDER BY`, `AS`.
All these operations are also (usually under a different name) present in the data manipulation libraries as well. We go through these operations and discuss their counterparts in Pandas.

SELECT

The SELECT statement in SQL serves to select a specific subset of columns of a table. In Pandas the square-brackets operator is used for that purpose. As listing 2.6 shows, the operator can return a Series or a Dataframe depending on whether the argument is just a string or a list of strings.

```
dataframe = ... # initial dataframe

# Select a subset of columns (returns a dataframe)
subset = dataframe[["column1", "column3"]]

# Select one column (returns a Series)
```

```
column = dataframe["column2"]
```

Listing 2.6 Select in Pandas

Alternatively, the filter function can be also used for this purpose. The important information for us will be that the operation returns a Dataframe with a different column structure than the input Dataframe—it removes non-specified columns.

The SELECT statement in SQL can also do more than just selecting a subset of columns. It can also apply aggregating function when used with group by. This option is covered later when we talk about the GROUP BY operation.

WHERE

The WHERE statement in SQL filters out the rows that do not match a given predicate. In this case, Pandas was also able to make use of the square-brackets operator. This time the operator accepts a Series of bool values and uses just the columns of a Dataframe with index the same as some True value in the Series. The Series of bools is usually (but not necessarily) created using vectorized operations. The listing 2.7 shows the usage.

```
dataframe = ... # initial dataframe

# Select just rows where the "age" column is at least 18
adults = dataframe[dataframe["age"] >= 18]
```

Listing 2.7 Where in Pandas

The important information for us will be that the result of this operation is a Dataframe with the same column structure.

AS

The AS keyword is used to rename a specific column. Pandas also has this feature, although it is named differently. The function is called rename, and it accepts a mapping from old column names to new column names. The listing 2.8 shows how the rename function can be used.

```
dataframe = ... # initial dataframe

# Rename some of the columns
renamed_dataframe = dataframe.rename(
    columns={"column1": "col1", "column2": "col2"}
)
```

Listing 2.8 As in Pandas

ORDER BY

ORDER BY is used to sort the data by some columns. Pandas can do the same using the sort_values function. The example usage can be seen in the listing 2.9

```
dataframe = ... # initial dataframe

# Order the dataframe rows by the values of column
```

```
# 'surname' in the ascending order
sorted = dataframe.sort_values(by=["surname"], ascending=True)
```

Listing 2.9 Order by in Pandas

Note that the operation does not change the column structure of the Dataframe.

JOIN

Join operation is used to combine rows from two tables based on some related columns. There are four types of join—inner, left, right and outer. Inner join returns rows that have matching rows in both tables. Left join returns all rows from the left table and all matched rows from the right table. Right join returns all rows from the right table and all matched rows from the left table. And outer join returns rows where there is a match in either of the tables.

Pandas has a function called merge for this purpose. It accepts two Dataframes and returns their corresponding join. Besides the already mentioned joins, merge also supports a cross-join, which is essentially a cartesian product of two sets of rows. The listing 2.10 shows how the merge function can be used as well as what parameters does the function accept.

```
dataframe1 = ... # 1st initial dataframe
dataframe2 = ... # 2nd initial dataframe

inner = pd.merge(
    dataframe1, dataframe2, how="inner",
    left_on="left_key", right_on="right_key")
left = pd.merge(
    dataframe1, dataframe2, how="left", on="common_key")
right = pd.merge(
    dataframe1, dataframe2, how="right", on="common_key")
outer = pd.merge(
    dataframe1, dataframe2, how="outer",
    left_on="left_key", right_on="right_key"
cross = pd.merge(
    dataframe1, dataframe2, how="cross")
```

Listing 2.10 Join in Pandas

GROUP BY

The group by operation aggregates the rows based on specified columns. It does not return normal rows but summary rows, which we can apply aggregate functions on. A Good example of an aggregate function can be max, min or sum.

Pandas has a function with the same name—groupby. The function returns a DataframeGroupBy object, which we can apply aggregate functions on. The listing 2.11 shows the usage

```
dataframe = ... # initial dataframe

# Group the dataframe by agg_column and return
# the mean of the "salary" column
dataframe.groupby(by=["work_department"])["salary"].mean()
```

Listing 2.11 Group by in Pandas

HAVING

Having operation works somewhat like a where operation on a summary rows—where group by was already applied. In having clause, we can use any aggregate function in a predicate and then filter the summary rows based on such predicate.

In Pandas, this can be done in many ways, but the usual one involves the filter function and lambdas. Example of such usage is shown in the listing 2.12.

```
dataframe = ... # initial dataframe

# Group the dataframe by agg_column and return
# the columns that are in a work_department
# with an average salary higher than 10 000
dataframe \
    .groupby(by=["work_department"]) \
    .filter(lambda x: x["salary"].mean() > 10000)
```

Listing 2.12 Having in Pandas

2.2.2 Vectorized operations

When we discussed the SQL WHERE clause, we came across the following line of code:

`adults = dataframe[dataframe["age"] >= 18]`

We said that this line of code filters out the rows where the age is less than 18. We also said that the square-bracket operator accepts a Series of bools. So the expression `dataframe["age"] >= 18]` must return a Series of bools. But why does this work? How can we compare a Series to a number?

In Pandas, the Series can be summed, subtracted, multiplied or compared with values that are either Series or a scalar values. The listing 2.13 shows the behavior of some vectorized operations in Pandas performed in the interactive mode.

```
>>> sr1 = pd.Series([1, 2, 3])
>>> sr1 + 7
0     8
1     9
2    10
dtype: int64
>>> 3 * sr1
0    3
1    6
2    9
dtype: int64
>>> sr1 / sr1
0    1.0
1    1.0
2    1.0
dtype: float64
>>> sr2 = pd.Series(["a", "b", "c"])
>>> sr2 * 2
0    aa
1    bb
2    cc
dtype: object
```

```
>>> sr2 + "x"
0     ax
1     bx
2     cx
dtype: object
```

Listing 2.13 Vectorized operations

Summary

We covered two common data structure interfaces used in data analysis - Series and Dataframe. Series is a one-dimensional List-like data structure with support for index and an axis label. Data frame is a two-dimensional structure—a dictionary of columns. The whole Dataframe can have an index associated, and the column of a Dataframe can be seen as a Series. Many operations in the data manipulation libraries are influenced by the SQL language operations such as join, select, group by, where, order by etc. The data structures also support vectorized operations like sums, products or comparing.

All discussed topics were demonstrated on Pandas code snippets. However, the Pandas library is a large project and the goal of this chapter was not to explore it all. More detailed information can be found in the API reference of Pandas[1].

3 Putting it all together

As the name of this chapter suggests, we now put the knowledge from the previous chapters together. We build the Abstract Interpretation framework for data manipulation programs. It involves defining the concrete and abstract lattice and the Galois connection between.

Since we are mostly focusing on Pandas in Python, we assume Python environment with Pandas, specifically:

- Python syntax
- Standard Python data types—int, float, string, list, dictionary, tuple
- Pandas is imported via the `import pandas as pd` statement

We also assume, unlike the Pandas, that Series are homogenous and Dataframes have homogenous columns. This also implies that each column of a Dataframe has an associated type.

Definition 14. *Let Df be a Dataframe and Sr a Series. We assume the following operations for accessing metadata about the two structures:*

```
columns(Df) = { Sr1 .. Srn }

type(Sr) - returns type

name(Sr) - returns name
```

Given these assumptions, we can define a Dataframe Structure:

Definition 15 (Dataframe Structure)**.** *For a Dataframe Df, its* ***Dataframe*** *Structure is $DS(Df) = \{(name(Sr), type(Sr)) | Sr \in columns(Df)\}$*

Also, we define a Series Structure:

Definition 16 (Series Structure)**.** *For a Series Sr, its* ***Series Structure*** *is $SS(Sr) = (name(Sr), type(Sr))$*

3.1 The concrete lattice

The concrete lattice should correspond with the reality. Since the variables of our program are Dataframes and Series, we can imagine the set of all possible Dataframes and Series (with every finite number of rows/columns and all possible values). Then we define the concrete lattice as the power set of this set with the inclusion being the ordering. To formalize this:

Definition 17 (The concrete lattice)**.** *Let*

$$C = \{\forall df \ Dataframe : df\} \cup \{\forall sr \ Series : sr\}$$

Then the concrete lattice is $L_c = (\mathbb{P}(C), \subseteq)$

3.2 The abstract lattice

The abstract lattice can be chosen by us depending on how much we want to abstract the values. We would like the abstraction to be actually useful in practice. What does this mean? In the Abstract Interpretation, whenever we should get a value from the user, we approximate it by the supremum of our lattice. This is usually done when we do not know anything about the input, since it is the most precise approximation that is still sound. However, we can do something smarter. We will create and use the assumption that whenever the user inputs a Dataframe, the Column Structure of that Dataframe is known in advance. This is not a very strong assumption in practice given the fact that the structure of the Dataframe is usually known during the process of writing the program anyway—otherwise we would not be able to write the program at all.

So the abstract lattice consists of the Dataframe Structures and Series Structures. We take the set of all possible Dataframe Structures and Series Structures. Then the abstract lattice is the power set of that set with the inclusion being the ordering. We again formalize this:

Definition 18 (The abstract lattice)**.** *Let*

$$A = \{\forall df\ Dataframe : DS(df)\} \cup \{\forall sr\ Series : SS(sr)\}$$

. Then the abstract lattice is $L_a = (\mathbb{P}(A), \subseteq)$

3.3 The Galois Connection

To create a Galois Connection, we need an abstraction function and a concretization function.

In the abstraction function, we start with a set of Dataframes and Series, and we want a set of Dataframe Structures and Series Structures. The most natural way to define such a function is to take the set of Dataframe Structures (Series Structures) of the Dataframes (Series) from the input. Formally:

Definition 19 (Abstraction function)**.** *Given the concrete lattice* L_c

$$\begin{gathered}
\forall X \in L_c \\
X_{df} = \{df \in X :\ df\ is\ Dataframe\} \\
X_{sr} = \{sr \in X :\ sr\ is\ Series\} \\
\alpha(X) = \{DS(df) \forall df \in X_{df}\} \cup \{SS(sr) \forall sr \in X_{sr})\}
\end{gathered}$$

Then we need the Concretization function. We will define it analogously. We have a set of Dataframe Structures and Series Structures on the input and want a set of Dataframes and Series on the output. So we just take all possible Dataframes (Series) with the given Dataframe (Series) Structure.

Definition 20 (Concretization function)**.** *Given the abstract lattice L_a*

$$\begin{gathered} \forall X \in L_a \\ X_{df} = \{df \in X : \ df \ is \ DataframeStructure\} \\ X_{sr} = \{sr \in X : \ sr \ is \ SeriesStructure\} \\ \alpha(X) = \bigcup_{dfs \in X_{df}} \{df : df \ is \ Dataframe \wedge DS(df) = dfs\} \cup \\ \bigcup_{srs \in X_{sr}} \{sr : sr \ is \ Series \wedge SS(sr) = srs\} \end{gathered}$$

3.4 Abstract Operations

In the first chapter, we mentioned that the abstract semantics can be systematically derived from the galois connection and the concrete semantics. However, it is often not the best way to get to the abstract semantics. We follow an alternative approach. We define the operations ourselves, since it is a very intuitive process. We do not define all the operations that we introduced in the previous chapter. We take a subset of them, and the rest can be found in the Pandalyzer implementation, and their formal model can be defined in a similar way as the operations discussed in this section.

We also define the operations on the single Dataframe (Series) Structures rather than on the sets, since it is easier to understand. It can then be extended to the variants with sets in a straightforward way.

- SELECT

  The first operation we will abstract is a simple SELECT. We are abstracting the selection of a subset of columns given their names. This is easy: We take our Dataframe Structure and remove all columns that do not have the names in the selection.

  Example. Input: DataFrameStructure(column1: int, column2, string, column3: bool), select([column1, column3])

  Output: DataFrameStructure(column1: int, column3: bool)

  In a real analysis, we should check that all column names specified in select exist in the DataFrameStructure and announce an error if the opposite is true.

- WHERE

  The WHERE operation has the nice property that it does not change the Dataframe Structure. So we just return the input DataframeStructure. In a real analysis, we should also check that the column referenced in the predicate exists in the DataFrameStructure and the comparison happens between compatible types—we should probably announce an error if we are trying to compare a number to a string.

- JOIN (merge)

The JOIN is more interesting. The input is two DataframeStructures, type of join and related column names. The output should be the DataframeStructure of the joined Dataframe. We should check that the columns that are used for joining exist in the original Dataframe and that the resulting Dataframe is valid—there are no duplicate columns.

- GROUP BY

  The GROUP BY operation does not return a Dataframe. It returns an object called DataFrameGroupBy. The object remembers the DataframeStructure and the column names that it was grouped by. When abstractly interpreting, we should also check that all the columns that we group by are actually present in the original Dataframe.

- Plus operator (vectorized)

  The Plus operator should work on a pair of two scalars, two Series and a pair of scalar and series. If only the types are known, we perform just the type check. If the values are both known, we perform the full operation.

3.5 Adding other types

So far, our framework only works with the Dataframe and Series types. Usually, however, these are not the only types in our program. Additionally, there are operations that accept Dataframe or Series and return other types. So we should also be able to work with normal ints, strings, lists, dictionaries or tuples.

We also abstract these types but in a slightly different way. For example, for a string, we do not remember some kind of abstract structure. Rather, we try to remember the whole string if it is possible. If it is not possible (we are not able to construct the string due to missing information), we just remember that the variable is a string with some unknown value (and we do not constraint the value in any way). This could be done better, but that is outside the scope of this thesis.

Ints or floats are handled in a similar way, but there is a difference in handling lists, dictionaries and tuples. We explain the concept on lists. In a situation when we are able to reconstruct the items of a list, we remember the whole list together with its items. If there is an item that we are not able to reconstruct, but we know that it is there, then we remember a marker structure called Unresolved Structure. And if we are not able to resolve the items of the list at all (not even their count), then we remember just the information that variable is a list (and we do not constraint the content of the list in any other way).

3.6 Final proposal

The only task left for us to finish the analysis framework is to combine the Dataframe and Series framework with the general abstraction proposed for other types (ints, strings, lists, ...).

We design a hierarchy of abstract types. The higher levels of the hierarchy contain less precise (more abstract) types, and the preciseness increases as we

go down the hierarchy. There is the Unresolved Structure at the bottom of the hierarchy representing a value that we were not able to derive. This is usually a result of an invalid operation or generally an invalid state. When we go up the hierarchy, the next level represents a different type of knowledge for different types; specifically:

- For elementary types (string, int, float) it is a complete knowledge of the value.
- For Dataframe and Series, it represents knowledge of the Dataframe Structure or Series Structure.
- For compound types (list, dictionary, tuple) we know their size, and for each of them we remember some level of abstraction from the hierarchy

When we go even one level higher, we abstract the concrete value to its type, meaning that we remember that the value is a string (int, list, Dataframe, . . .) but we do not know anything else about it. The idea is shown in the Figure 3.1. The Value means a specific value such as string literal or a number. The Content represents an abstract type in the hierarchy.

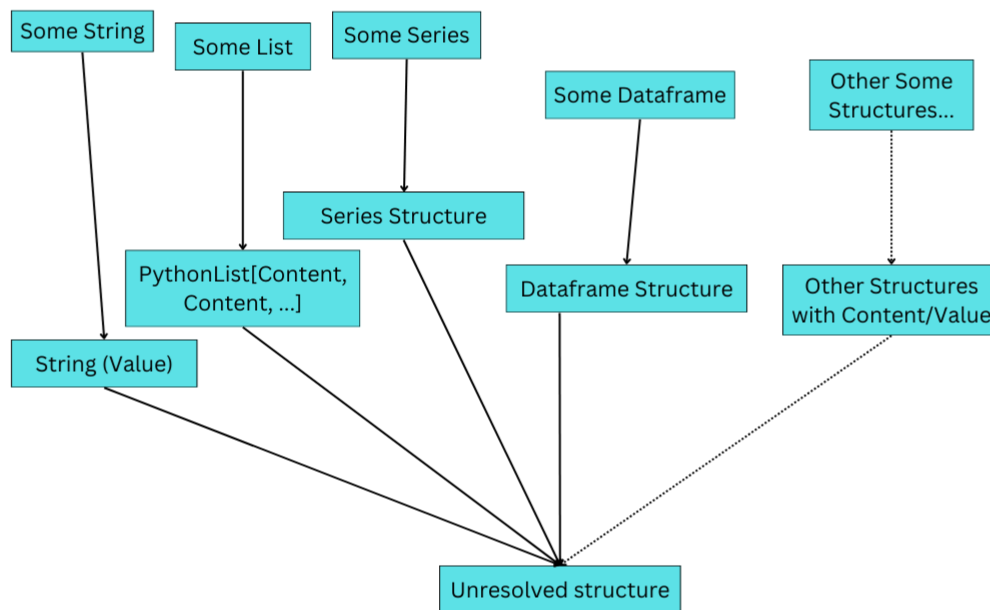

Figure 3.1 Abstract structures hierarchy

To satisfy the definition of a Lattice, every pair of elements must have a supremum and infimum. It can be seen from the Figure 3.1 that the infimum exists. For two different values of the same type `String("first")` and `String("second")`, the infimum is `SomeString`. For the two different types `String("str")` and `Int`7))|, the infimum is `UnresolvedStructure`. The other cases are analogous and are left as an exercise for the reader.

However, the idea misses suprema. For that purpose, we define another structure called `NondeterministicStructure`. The supremum of two elements of the hierarchy `a` and `b` is `NondeterministicStructure(a, b)`. The idea is also applicable recursively, so the value

```
NondeterministicStructure(
    NondeterministiStructure(
        SomeString,
        SomeInt
    ),
    SomeSeries
)
```

is a valid element of the hierarchy. The idea is shown in the Figure 3.2 With this idea, the hierarchy is finally a Lattice. Note that it is not a Complete Lattice as the whole hierarchy does not have a supremum.

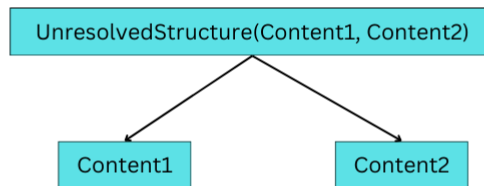

Figure 3.2 Nondeterministic structure

The Nondeterministic Structure will mostly occur when there is a branching (e.g. an if-statement) in the program, and we are not able to resolve which branch we should go. So the Nondeterministic Structure is a substitute that we need to use since we do not use the power set as the abstract lattice anymore.

3.7 Limitations

The presented framework has some limitations that are important to discuss. First, remembering just a Dataframe Structure prevents us from doing operations, that have the resulting Dataframe Structure dependent on the values in a Dataframe. An example of such operation can be the pivot operation. The result of a pivot operation has columns named after values from a selected column. But we do not know these values, since we only remember the Dataframe Structure. Another example of an operation, that we are unable to resolve the resulting Dataframe of is a transpose. The transpose function flips the whole Dataframe along the diagonal (like with matrices), so the columns of the result will be the rows of the original Dataframe, and the column names of the result will be the index values of the original. But, again, we do not know the index values. In such a situation, there is no other option than just returning Some Dataframe.

Another possible problem can be an unknown value propagation, meaning that when there is a value with any uncertainty (e.g., Nondeterministic Structure or Some value), using this value in an operation will in most cases lead to an operation result with an uncertainty as well.

Summary

We proposed a framework for analysis of a data-manipulation code. We showed what is the concrete lattice. We defined a hierarchy of structures forming the

abstract lattice. We also showed how the operations on Dataframes and Series work. All the knowledge from this chapter will be useful for us when implementing the data-manipulation code analyzer in the next chapter.

4 Pandalyzer

Our goal for this chapter is to present the implementation of the Pandalyzer, an analysis tool based on the Abstract Interpretation framework proposed in the previous chapter.

We start by stating the goal of the implementation—what features and functionality should the tool have. Then we present the architecture of the implemented solution from the software engineering point of view. We talk about some design decisions done and discuss their consequences. The User documentation can be found in the Appendix A.

4.1 The Goal

The implementation of an analyzer of a Pandas (Python) code based on Abstract Interpretation is a very broad assignment specification, so we need to set some scope limitations for our implementation. We do not want to support all Python language constructs and features as that would result in a full-blown Python interpreter implementation which is definitely out of scope for this thesis. On the other hand, our solution should be useful in practice, so the Python constructs that are often used in the context of data manipulation should be supported. This means that we definitely want to support assignments, function definitions and calling with return values and arguments, unary and binary operators, if-statements, constants and variables of various types. However, the implementation does not have to support classes, list comprehensions, lambdas, match statements, async code or slices in a subscript operator, although the code should be extensible enough so that these constructs can be added in the future development.

We also do not want to support all Pandas features as Pandas is a large project with very complex (and sometimes inconsistent) API. What we want to support are, again, the common features such as merging, grouping and aggregations, selection of subset columns, renaming of columns and creating DataFrames and Series from lists or dictionaries. We also want to support reading the DataFrames from a CSV or other file formats in some way. Less frequently used operations do not have to be supported, but the set of Pandas operations should be easily extensible with other operations.

The tool can be implemented as a command-line application, and a user should be able to build and run it on Linux, Windows, and macOS. It should accept a single Python script filename as an argument and should print the analysis result to standard output or a file. The output format should be configurable.

Another requirement is the ability to continue in the analysis when a mistake in the code is found and the ability to handle also some mistakes that are not related to Pandas but are just Python mistakes.

Pandas Dataframes are usually loaded from CSV files and also written to CSV files. The analyzer should not read or create any CSV files. It should accept information from the user about existing CSV files, and the analysis result should contain information about which CSV files would be created by the script and what would be their structure.

4.2 Architecture

The programming language chosen for the implementation is Kotlin [8]. The project uses Gradle [9] as a build system, and it runs on JVM [10]. The source code can be found in the Pandalyzer git repository [11].

4.2.1 The high-level idea

Let us go over what the tool does. It loads the Python script from the given input file. Then it parses the code and creates and abstract syntax tree (AST) of the module. Then it goes over the statements in the body of the module and interprets them while keeping the current context containing all currently active variables, raised errors and warnings, etc. Finally, it writes the result of the analysis to the standard output or to an output file (if provided).

The text above is a good short description of the tool, but it probably leaves the reader with many questions unresolved. In the rest of the chapter, we answer the following questions:

- How is the parsing done? (Section 4.2.2)
- How is the AST represented in the program? (Section 4.2.2)
- How are the currently active variables and other data represented in the context? (Section 4.2.3)
- How are the data structures represented (Sections 4.2.4 and 4.2.6)
- How are the Pandas operations and other functions checking implemented? (Section 4.2.7)
- How does the tool handle non-deterministic or unknown data? (Section 4.2.5)
- How does the tool accept the additional information about the input CSV files? (Section 4.2.8)

4.2.2 Design Decision - Parsing and AST

The tool needs to construct the abstract syntax tree of a Python module. Implementing a Python parser from scratch would be a lot of unnecessary work and a potential source of bugs, so we decided to use already existing solutions. There is a Python module called **ast** [12] capable of parsing a python module and creating an abstract syntax tree. However, since it is a Python module, we are not able to use it in Kotlin directly. We choose an alternative approach. We create a Python script that accepts a Python module, creates an abstract syntax tree using the **ast** module and serializes the tree to a json format. Then we use the Kotlin Serialization [13] library to deserialize the json to the kotlin representation. The Python script can be found in the Pandalyzer repo [11] in the path /src/main/resources/python_converter.py and uses serialization ideas from the ast2json Python module [14]. To avoid the need to run the converter script before running the Pandalyzer, the Pandalyzer calls the script instead of the user, effectively hiding the information about the existence of the script from

the user. However, the fact that we use a Python script implies that the python needs to be installed on the machine.

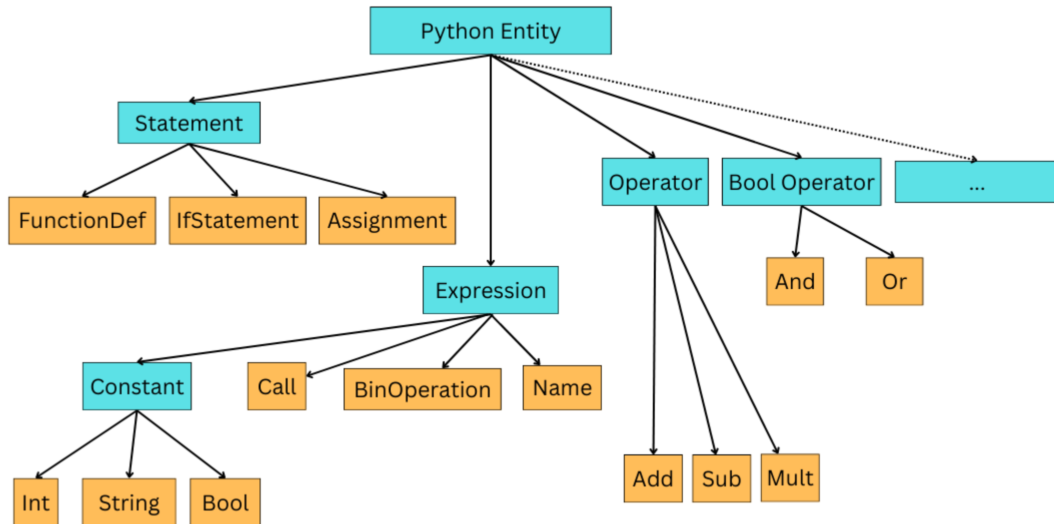

Figure 4.1 AST representation hierarchy

The AST representation in Kotlin uses hierarchy of classes and interfaces. The figure 4.1 shows a subset (the whole hierarchy is much larger) of the classes (yellow boxes), and interfaces (blue boxes).

Each class represents a node in the abstract syntax tree. The nodes of the tree have children depending on their semantics. For example, the node Module has a list of Statement nodes representing the body of the module. Another good example is an IfStatement, which has test Expression as a child, list of statements as the body of the if-statement and another list of statements as the body of the else branch.

Additionally, each node implementing the Statement interface has also information about the location of the statement in the script file - line numbers and column numbers.

4.2.3 Analysis Context

When traversing the AST and interpreting the program, we have to keep track of all currently known variables. We also need to store the errors and warnings generated. We introduce the Analysis Context structure for this purpose. In the implementation, AnalysisContext is an interface providing us with function such as getStruct, upsertStruct, addWarning, addError, and more. There are two classes implementing this interface—GlobalAnalysisContext and FunctionAnalysisContext.

GlobalAnalysisContext is the context that keeps track of all global variables and functions and Python builtin functions as well. It also contains metadata about the CSV files that we read from and write to.

FunctionAnalysisContext on the other hand is created when a function is invoked. It contains a reference to outer analysis context and a global analysis context. It redirects the addWarning and addError function to the outer analysis context, and it also redirects the getStruct function to the global analysis context

in case that it does not know the requested variable. This ensures that the concept of global and local variables works properly.

When the analysis is done, the global analysis context has a `summarize` function, which returns a summary of the analysis.

4.2.4 Python structures representation

When we discussed the Analysis Context, we said that it keeps all currently active variables. We use polymorphism to keep track of variables with a dynamically resolved type. We define the interface PythonDataStructure. Every Python data type has to implement the PythonDataStructure interface. The interface contains methods representing what we can do with any Python data type. There are functions for the implementation of binary operators such as add, subtract, multiply, etc., unary operators such as unary plus or unary minus, subscript function (the square bracket operator), the invoke function (the parenthesis operator) and the attribute function (used for the dot notation).

All these functions return the following type:

```
OperationResult<PythonDataStructure>
```

So the operations return a PythonDataStructure, but it is wrapped inside the OperationResult. The OperationResult is a sealed interface (meaning that all who implement the interface must be known at compile time). There are three classes implementing the OperationResult interface: OK, Warning and Error. OK signalizes that the operation succeeded, and it contains the result PythonDataStructure. Warning tells us that the operation was successful but some warnings were raised during the execution. The Warning type contains the result PythonDataStructure and a list of warnings - strings. Last type is the Error. It tells us that the operation failed, and it also provides a reason string.

We said that every Python data type has to implement the PythonDataStructure interface. We would like to define abstract versions of standard Python types, meaning PythonString, PythonInt, PythonBool, PythonNone, PythonList or PythonDict. There are also functions (yes, a function is also a PythonDataStructure). This includes (subset of) Python builtin functions such as `print, list, len, int, or abs` as well as other functions defined in the current scope. Another thing that also implements the PythonDataStructure is imported modules. Currently, the only supported import is PandasImport, but the implementation can be easily extended by others. The Pandas data types support is added by creating DataFrame and Series classes that also implement the PythonDataStructure interface.

4.2.5 Nondeterminism, unknown values and error recovery

There are three ways in which we handle uncertainty. The first one corresponds to the SomeString/SomeList/...structures mentioned in the previous chapter. Recall that these structures represent the knowledge of the type but the absence of the knowledge of the value. In the implementation of Pandalyzer, this concept is represented by the fact that the values inside PythonString, PythonList, ... are nullable. In other words, the PythonString(value=null) represents a string with unknown value.

The second form of uncertainty is the Nondeterministic data structure (also defined in the previous chapter). This is the uncertainty between two possible values. In the implementation of Pandalyzer, this is represented by NondeterministicStructure with left and right PythonDataStructure. The Nondeterministic-Structure(left=PythonList([]), PythonInt(null)) represents a value that is either an empty list or a some unknown integer. When we perform an operation on this structure, it internally performs the operation on both options and then returns both results wrapped in a new NondeterministicStructure.

The last form of uncertainty is the UnresolvedStructure (also defined in the previous chapter). It simply means that we do not know at all what the value is. It happens when there is an error, and it is a part of the error recovery mechanism of the analyzer. When we try to do an operation that is not permitted, the result of that operation is the UnresolvedStructure with reason inside. The reason is then added to the list of all errors.

The Pandalyzer is able to execute multiple branches of if-statement if it is not able to resolve, which branch it should choose. Each PythonDataStructure implements the boolValue function. The boolValue function returns True, False or null. If the boolValue of the expression in the condition of the if-statement returns null, we have to execute both the branches of the if-statement. For this purpose, we introduce the fork-join pattern on the Analysis Context. The Analysis Context has fork and join functions. The fork function returns a new Analysis Context that is equal to the original Analysis Context. Then we use the original context to execute the body of the if-statement, and we use the forked context to execute the else branch of the if-statement. Then we call `originalContext.join(forkedContext)`, which merges the forked context back to the original context. If some of the values differ across the contexts, we create a Nondeterministic Structure and insert the different structures there. We also combine the warnings and errors of the two contexts.

4.2.6 Dataframe and Series Representation

The Dataframe and Series representation mostly correspond to the Dataframe Structure and Series Structure defined in the previous chapter. The listing 4.1 shows the definition of DataFrame in the Pandalyzer source code.

```
data class DataFrame(
    val fields: MutableMap<FieldName, FieldType>?,
) : PythonDataStructure
```

Listing 4.1 Kotlin definition of the Dataframe

The DataFrame class implements the PythonDataStructure interface since it is a type in Python. It has a single (mutable) map, where the key type is a FieldName, which is a string. The FieldType is an enum of all possible types that a column of a Dataframe in Pandas can have. Also note that the map is nullable. The `DataFrame(fields=null)` represents a Dataframe with unknown Dataframe Structure.

The listing 4.2 shows the definition of Series in the Pandalyzer source code.

```
data class Series(
    val label: FieldName?,
    val type: FieldType?,
```

```
) : PythonDataStructure
```

Listing 4.2 Kotlin definition of the Series

The Series class also implements the PythonDataStructure interface. The Series keeps the type information as well as the label. Note that both the fields are nullable, meaning that it can happen that we do not know them.

Other Pandas types implemented in the Pandalyzer are SeriesGroupBy and DataframeGroupBy—the results of the groupby operation.

4.2.7 The Abstract Operations

When implementing the abstract operations, we have to perform many checks before the abstract operation itself. We have to validate all the arguments—they can be nondeterministic, they can have null value inside, or, if they are a compound type (list or dictionary), they can have an element that has a null value inside.

We present the idea of the implementation on an example. The example is the rename function of a Dataframe. It (in its simplest form) accepts an argument containing a dictionary mapping old column names to new column names. The rename call can look as follows: `df.rename({"old1": "new1", "old2": "new2"})`.

The `df.rename` says that we are accessing `rename` attribute of the `df` variable. The implementation of `attribute` function of DataFrame is shown in the listing 4.3

```
override fun attribute(identifier: Identifier
        ): OperationResult<PythonDataStructure> =
    when (identifier) {
        "rename" -> DataFrame_RenameFunc(this).ok()
        "merge" -> DataFrame_MergeFunc(this).ok()
        \* snip... other functions *\
        else -> fail(
            "Unknown identifier on dataframe: $identifier"
        )
    }
```

Listing 4.3 Snippet of attribute function of Dataframe

The attribute function accepts the `identifier` argument and checks if it is the name of one of the known functions. If it finds the right function, it returns the object representing the function (wrapped in an OperationResult).

Then the part `rename(...)` means that we call invoke on the DataFrame_RenameFunc object. The invoke function parses the argument, checks that it is a PythonDict, does non-deterministic split if necessary and calls the rename function which performs the logic of the rename operation. The invoke function is rather technical, so we skip that. Let us instead discover what the rename function of DataFrame_RenameFunc object looks like. It can be seen in the listing 4.4.

```
private fun rename(dict: PythonDict
    ): OperationResult<PythonDataStructure> {
    if (dict.values == null) {
        return DataFrame(null)
            .withWarn("Unable to rename dataframe
                because the values of dictionary are unkown")
    }
    if (dataFrame.fields == null) {
```

```
        return DataFrame(null)
            .withWarn("Unable to rename
                dataframe with unknown structure")
    }

// check that the old values in the mapping dict are all strings
    val nonStringOldValues = dict.values
        .filterKeys { (it is PythonString).not() }.keys
    if (nonStringOldValues.isNotEmpty()) {
        return fail("The old column names
            should be strings, but were $nonStringOldValues")
    }

// check that the new values in the mapping dict are all strings
    val nonStringNewValues = dict.values
        .filterValues { (it is PythonString).not() }.values
    if (nonStringNewValues.isNotEmpty()) {
        return fail("the new column names
            should be strings, but were $nonStringNewValues")
    }

// check that the old and new values in the mapping are known
    val nullMapping = dict.values.map {
        (it.key as PythonString).value
            to (it.value as PythonString).value
    }
    if (nullMapping.any {
        it.first == null || it.second == null }){
        return DataFrame(null)
            .withWarn("Unable to resolve
            some mapping parts of rename function")
    }
    val mapping = nullMapping
        .associate { it.first!! to it.second!!}

    // check that all the old values exist in the dataframe
    val missingOldValues = mapping.keys
        .filterNot { it in dataFrame.fields }
    if (missingOldValues.isNotEmpty()) {
        return fail("The values $missingOldValues
            do not exist in the dataframe")
    }

// check that new values are not colliding with old values
    val collidingValues = mapping
        .filter { it.value in dataFrame.fields }
    if (collidingValues.isNotEmpty()) {
        val message =
            collidingValues
                .map {
            "Cannot rename a dataframe column
            ${it.key} to ${it.value} " +
            "since ${it.value} already
            exists in the dataframe"
                }
                .joinToString("\n")
        return fail(message)
    }
```

```
    // check that there are no duplicate new values
    val duplicateValues = mapping.values
        .groupBy { it }.filterValues { it.size > 1 }
    if (duplicateValues.isNotEmpty()) {
        return fail("There are duplicate new values in
            the rename function: ${duplicateValues.keys}")
    }

    return DataFrame(fields = dataFrame.fields
        .mapKeys { mapping[it.key] ?: it.key }.toMutableMap()
    ).ok()
}
```

Listing 4.4 The rename function of DataFrame_RenameFunc

Note that the example was altered to fit horizontally on the page. The rename function first checks that no value needed to determine the result is null. It also checks that both the keys and values are all strings, since that is the only accepted option for a rename function. The next check ensures that the keys in the dictionary have the corresponding columns in the Dataframe. Last thing that the function checks are collisions and duplicates. Then, new Dataframe with the new column structure is returned. Note that we fail only if there is something wrong. We do not fail if we do not have enough information. Rather, we return a Dataframe with unknown structure and also add a warning.

4.2.8 Design Decision - Configuration file

When we defined the goals of the implementation, we mentioned the fact that Pandas Dataframes are usually loaded from a CSV file. We stated a requirement that the analyzer should not try to read the file. The analyzer should instead receive the information about the CSV file structures from the user in some other way. This has the advantage that the CSV files do not have to exist at all yet during the analysis as long as we know their structure.

This, however, raises a question: How should the Pandalyzer receive the information from the user? The following options were considered:

- The user specifies the CSV structures as a command-line arguments

  This was not accepted since it is not practical for the case when the set of CSV files is bigger or the CSV files have many columns.

- The user adds the information straight to the Python code via some annotation or nop operation

  This was also rejected since the user has to make changes to the code, and it could make the code less readable.

- The tool will ask the user for the information once it recognizes the read_csv function

  The problem with this approach is that it needs a lot of interaction with the user, and it complicates automatic execution of the analysis.

- The user will provide a configuration file with the CSV structures

  This approach has the advantage that it is relatively easy to specify many files that have many columns. Also, the configuration can be written just once in a whole project and be a part of the repository. So we chose this approach in the implementation.

The format of the configuration file can be best explained by an example.

```
[file.csv]
column1 = "string"
column2 = "int"

[file2.csv]
columnA = "int"
columnB = "int"
columnC = "bool"
```

Listing 4.5 An example configuration file

As seen in the Listing 4.5, the format is inspired by a well-known configuration format TOML [15]. We chose TOML because of its excellent readability. However, we do not use the TOML language but only a small modified subset. The definition of each file structure begins with a line containing the (relative or absolute) path to the file in the file system surrounded by square brackets. Note that this is already not a TOML, since the path to file can also contain dots or slashes which are not permitted in TOML specification. The following lines contain the specification of columns starting with column name, then equal sign and then the type of the column surrounded by double quotes.

Another feature of the configuration is the regex-based file definition. When we prepend the line containing the definition of new file structure with the letter `r`, the filename is considered a regex pattern and each file matching the pattern will have the defined structure. The idea is shown in the Listing 4.6.

```
r[^\d\d_file\.csv]
col1 = "string"
col2 = "int"
```

Listing 4.6 An example regex-based file definition

This means that files with names such as 01_file.csv, 02_file.csv, 99_file.csv have one column of type string.

Summary

We presented the Pandalyzer—a data manipulation code analyzer for Python and Pandas. The tool loads the Python code, then parses it and converts it to the AST using a Python script. Then the AST is traversed and partially interpreted. We keep the Analysis Context with all variables, warnings and errors. The data structures are represented by a set of classes implementing the PythonDataStructure. The uncertainty is handled using the NondeterministicDataStructure, UnresolvedDatastructure and null values inside the typed structures. The Pandas structures—Dataframe and Series—correspond to the Dataframe Structure and Series Structure defined in the previous chapter. The operations implement a lot

of checks that should prevent the user from doing many common mistakes. The Pandalyzer also supports config file, where the user can specify the structure of the input csv files. The planned future extensions are mentioned in the Future Work 5.1 section of the Conclusion.

5 Evaluation of the solution

This chapter is dedicated to the evaluation of the implementation presented in the previous chapter. We design a set of realistic case studies for Pandas and evaluate the quality of analysis of the proposed solution. We do not define any specific metrics for the evaluation since it is cumbersome to do it rigidly. We rather look at the analysis from the more intuitive perspective and discuss which useful analysis features are provided and which could be missing.

5.1 Case Studies

Each case study contains:

- An explanation of the case
- An example code in Python
- Description of possible mistakes
- The output of the analyzer in the good case and in the case with mistakes

Case Study #1: Multiple operations, multiple Dataframes, no uncertainty

The purpose of the first case study is to show the capabilities of Pandalyzer in a deterministic environment when Pandalyzer has all the information it needs for the analysis, and there is no uncertainty. There are, however, many different operations with multiple Dataframes. We show that Pandalyzer is able to check input for all these operations and determine their output.

In this case study, we work with data from a sport competitions agency. We get a CSV file with competitions that they organized. We also get a CSV file with all attendees of these competitions. The last file contains information about the number of points that an attendee got from a specific competition.

The `config.toml` (and the CSV column structure) is shown in the listing 5.1

```
[attendees.csv]
name = "string"
surname = "string"
age = "int"

[matches.csv]
id = "int"
name = "string"

[scores.csv]
name_surname = "string"
match_id = "int"
score = "int"
```

Listing 5.1 config.toml of the first case study

Our goal is to create a CSV file called top_two_per_age.csv that contains the top two attendees per age per sport match together with the sport match name.

The listing 5.2 shows how that could be implemented in Pandas.

```
1  import pandas as pd
2
3  attendees_df = pd.read_csv("attendees.csv")
4  matches_df = pd.read_csv("matches.csv") \
5      .rename(columns={"name": "match_name"})
6  scores_df = pd.read_csv("scores.csv")
7
8  attendees_df["name_surname"] = \
9      attendees_df["name"] + "_" + attendees_df["surname"]
10 attendees_df = attendees_df.drop(columns=["name", "surname"])
11
12 scores_with_match_name_df = scores_df \
13     .merge(matches_df, left_on="match_id", right_on="id") \
14     .drop(columns="id")
15
16 scores_with_age_df = pd.merge(
17     scores_with_match_name_df, attendees_df, on="name_surname"
18 )
19
20 top_two_per_age_df = scores_with_age_df \
21     .sort_values("age") \
22     .groupby(["age", "match_name"]) \
23     .head(2) \
24     .drop(columns=["match_id"])
25
26 top_two_per_age_df.to_csv("top_two_per_age.csv")
```

Listing 5.2 Solution of the first case study in Pandas

Let us break down the code in the listing 5.2. We first read all the CSV files and store the data to Pandas Dataframes. We also rename the `name` column in `matches_df` to `match_name` so that the name of the column makes sense later when it is merged with other Dataframes. This is the first tricky part; since from now on we have to remember that the `matches_df` does not contain a name column but rather a `match_name` column. Then, since `attendees_df` contains `name` and `surname` columns and the `scores_df` contains `name_surname`, we also have to create a new column `name_surname` in the `attendees_df` and drop the old columns (`name` and `surname`). The next two operations merge the three Dataframes to a `scores_with_age_df`. We can notice that it is now already hard to keep track of the structure of the Dataframes. Then the Dataframe is sorted by `age`, grouped by `age` and `match_name` and the first two items from each group are selected. Also, we drop the `match_id` since we do not need it in the final data. The resulting Dataframe is stored to a CSV file called `top_two_per_age.csv`.

This case study shows two ideas. First, it is hard to keep track of everything what is happening with the Dataframes, and it is easy to do a mistake e.g. access an already non-existent column, misspell the column name or incorrectly specify the arguments of a Pandas function. Second, the scenario where we are able to deterministically derive the structure of the resulting data (e.g. it does not depend on any user input or other sources of uncertainty) can be a very common scenario.

What we want from the Pandalyzer in this case is to give us the structure of the `top_two_per_age.csv` file and warn us in case that we do any of the mistakes mentioned. The listing 5.3 shows the output of the Pandalyzer given the input shown in the listing 5.2 and the configuration shown in the listing 5.1.

```
Summary of analysis: OK
```

```
Global data structures (7):
pd: PandasImport
attendees_df: DataFrame(columns={age=IntType,
    name_surname=StringType})
matches_df: DataFrame(columns={id=IntType,
    match_name=StringType})
scores_df: DataFrame(columns={name_surname=StringType,
    match_id=IntType, score=IntType})
scores_with_match_name_df: DataFrame(columns={
    name_surname=StringType, match_id=IntType,
    score=IntType, match_name=StringType})
scores_with_age_df: DataFrame(columns={name_surname=StringType,
    match_id=IntType, score=IntType,
    match_name=StringType, age=IntType})
top_two_per_age_df: DataFrame(columns={name_surname=StringType,
    score=IntType, match_name=StringType, age=IntType})

Warnings (0):

Errors (0):

Output files (1):
File top_two_per_age.csv:
    name_surname : StringType
    score : IntType
    match_name : StringType
    age : IntType
```

Listing 5.3 Output of Pandalyzer on the first case study

The first line tells us that the no mistake was spotted in the analyzed code (otherwise there would be `NOT OK`). Then, the analyzer tells us all variables that are in the global scope when the analysis ended. We can see all the Dataframes with their structure. Next, we can see that the lists of Warnings and Errors are empty. Finally, there is the `Output files` summary with the `top_two_per_age.csv` file structure. The analyzer is able to tell us what columns the file contains as well as what are their types.

Now we show what happens when we do some mistake in the code.

The listing 5.4 shows the changed line 5 (`neme` instead of `name`) in the code as well as the analysis output.

```
    .rename(columns={"neme": "match_name"})

Summary of analysis: NOT OK
Global data structures (7):
pd: PandasImport
attendees_df: DataFrame(columns={age=IntType,
    name_surname=StringType})
matches_df: UnresolvedStructure(reason=The values
    [neme] do not exist in the dataframe)
scores_df: DataFrame(columns={name_surname=StringType,
    match_id=IntType, score=IntType})
scores_with_match_name_df: UnresolvedStructure(
    reason=Incorrect right argument to merge)
scores_with_age_df: UnresolvedStructure(reason=
    Incorrect left argument to merge)
top_two_per_age_df: UnresolvedStructure(reason=
    the attribute sort_values of UnresolvedStructure
```

```
    does not exist)

Warnings (0):

Errors (5):
0: Assign from line 4 to line 5 columns 0 - 44:
    The columns [neme] do not exist in the dataframe
1: Assign from line 12 to line 14 columns 0 - 23:
    Incorrect right argument to merge
2: Assign from line 16 to line 18 columns 0 - 1:
    Incorrect left argument to merge
3: Assign from line 20 to line 24 columns 0 - 31:
    the attribute sort_values of UnresolvedStructure
    does not exist
4: ExpressionStatement on line 26 columns 0 - 48:
    the attribute to_csv of UnresolvedStructure does not exist

Output files (0):
```

Listing 5.4 Misspelled column on rename function and analysis output

Now we can see that the result of the analysis is `NOT OK`. There are five errors in the Errors list. The first error tells us that in the statement on lines 4–5 there is a statement where we are trying to access column `neme` that does not exist in the Dataframe. This is enough for us to spot and fix the mistake in the code. Other errors are in this case just consequences of the first error.

Another mistake that we can do is to specify an incorrect argument to a Pandas function. The listing 5.5 shows the changed part of the code on line 13 (`left_or` instead of `left_on`), and the analysis output. Note that only important parts of the analysis output are shown.

```
    .merge(matches_df, left_or="match_id", right_on="id") \

Summary of analysis: NOT OK
/* snip */
Errors (4):
0: Assign from line 12 to line 14 columns 0 - 23:
    Got unexpected keyword arguments [left_or]
/* snip */
```

Listing 5.5 Incorrectly specified argument and analysis output

The result of the analysis is `NOT OK` as expected, and there is an error saying that there was an unexpected keyword argument `left_or` on the lines 12–14.

Case Study #2: Uncertainty from the user, multiple possible values

Next case study shows how the Pandalyzer behaves when there is some uncertainty such as an input from the user. It also proves that the Pandalyzer is able to handle non-trivial control flow such as if-statements, functions, and early returns.

The code for this case study can be seen in the listing 5.6.

```
import pandas as pd


def get_country_dataframe(country):
```

```
    if country == "Germany":
        return pd.read_csv("de.csv")
    elif country == "Austria":
        return pd.read_csv("au.csv")
    else:
        return pd.read_csv("world.csv")


def get_dataframe_from_user():
    country = input("Select a country: ")
    return get_country_dataframe(country)


user_df = get_dataframe_from_user()
user_df[["germany_specific_column"]].to_csv("output.csv")
```

Listing 5.6 Code of the second case study in Pandas

And the configuration file is shown in the listing 5.7.

```
[de.csv]
germany_specific_column = "string"
common_column = "int"

[au.csv]
common_column = "int"

[world.csv]
common_column = "int"
```

Listing 5.7 config.toml file for the second case study

There are two functions. The first (`get_country_dataframe`) returns the Dataframe for the specified country or the Dataframe for the whole world if the country is not known. The second function (`get_dataframe_from_user`) gets the country from the user and returns the result of the `get_country_dataframe` function. However, we do not know the result of the `input` function, so we are not able to resolve which Dataframe will be returned. Moreover, we are trying to access the column `germany_specific_column` which exists only if the user inputs the value `Germany`. So the program can (but does not have to) crash.

The listing 5.8 shows the output of the Pandalyzer given the input from the listing 5.6 and the configuration file from listing 5.7.

```
Summary of analysis: OK
/* snip */
Warnings (6):
0: Assign on line 14 columns 4 - 41:
    Unable to resolve result of input
1: IfStatement from line 5 to line 10 columns 4 - 39:
    Unable to check unknown strings on equality
2: IfStatement from line 5 to line 10 columns 4 - 39:
    Unable to recognize the bool value in the
    if statement test - branching.
3: IfStatement from line 7 to line 10 columns 4 - 39:
    Unable to check unknown strings on equality
4: IfStatement from line 7 to line 10 columns 4 - 39:
    Unable to recognize the bool value in the
    if statement test - branching.
5: ExpressionStatement on line 19 columns 0 - 57:
```

```
    Second branch of execution failed with reason:
        Both execution branches failed.
        Branch1:
            The keys [germany_specific_column] do not
            exist in the dataframe,
        Branch2:
            The keys [germany_specific_column] do not
            exist in the dataframe

Errors (0):

Output files (1):
File output.csv:
    germany_specific_column : StringType
```

Listing 5.8 Analysis output of the second case study

This time the output is more complicated. The result of the analysis is `OK` since the fact that some branch can fail does not mean that the whole program is incorrect. However, there are five warnings. The first warnings is related to the `input` function itself. It just tells the user that an uncertainty occurred. The next four warnings are related to the fact that we are trying to compare a string with unknown value and then trying to branch based on the result. The Pandalyzer tells us that it is branching (executing both branches) in that case. The last warning is the most important. It is less readable, but that is just because the problem is complicated. It tells us that one of the branches failed, and it gives us the reason. The reason is that both branches in the second branch failed, and it gives us the reasons. The Pandalyzer also tells us that there is an output file. It corresponds to the branch of execution in which the user input was `Germany`.

Case Study #3: Regex config, column compatibility

Now we have a set of files 30_04_2024_production.csv and 31_04_2024_production.csv that contain per-hour production of some factory on a given day. We are interested in the hours when the production was lower than 400 items. This time we use the regex feature of our configuration. The `config.toml` file will be as shown in listing 5.9.

```
r[^\d{2}_\d{2}_\d{4}_production\.csv$]
hour = "int"
production = "int"
note = "string"
```

Listing 5.9 config.toml of the second case study

The record in the `config.toml` file says that all the CSV files with the name dd_mm_yyyy_production.csv have the specified columns structure. The regex feature is useful for large number of same-structured files with similar names (different only in date, number, etc.).

The code solving this problem can be seen in Listing 5.10

```
1 import pandas as pd
2
3 tuesday_df = pd.read_csv("30_04_2024_production.csv")
4 wednesday_df = pd.read_csv("31_04_2024_production.csv")
5
```

```
6  tuesday_df["day"] = 30
7  wednesday_df["day"] = 31
8
9  agg_df = pd.concat([tuesday_df, wednesday_df])
10
11 low_production_df = agg_df[agg_df["production"] < 400]
12
13 low_production_df.to_csv("aggregate_production.csv")
```

Listing 5.10 Solution of the third case study in Pandas

The output of the Pandalyzer given the input from listing 5.10 and the configuration from listing 5.9 is shown in the listing 5.11

```
Summary of analysis: OK
Global data structures (5):
pd: PandasImport
tuesday_df: DataFrame(columns={hour=IntType,
    production=IntType, note=StringType, day=IntType})
wednesday_df: DataFrame(columns={hour=IntType,
    production=IntType, note=StringType, day=IntType})
agg_df: DataFrame(columns={hour=IntType,
    production=IntType, note=StringType, day=IntType})
low_production_df: DataFrame(columns={hour=IntType,
    production=IntType, note=StringType, day=IntType})

Warnings (0):

Errors (0):

Output files (1):
File aggregate_production.csv:
    hour : IntType
    production : IntType
    note : StringType
    day : IntType
```

Listing 5.11 Analysis output of the third case study

As expected, the result of the analysis is `OK` and the lists of Warnings and Errors are empty. There is one output file `aggregate_production.csv` with the correct structure. The Pandalyzer was also able to match the input filenames with the regex in the configuration file. An important part of the analysis is the `concat` function. The result of the function has the same structure as the inputs, but it requires the inputs to have the same structure. Let us see what the analyzer outputs if we change the structure of one of the Dataframes. The listing 5.12 shows the changed line 6—we add a different column to one of the Dataframes, and the important parts of the analysis output.

```
tuesday_df["another_column"] = 30

Summary of analysis: NOT OK
/* snip */
Errors (3):
0: Assign on line 9 columns 0 - 46:
    All dataframes to be concatenated must have
    the same column structure
/* snip */
```

Listing 5.12 Incompatible Dataframes to concat operation and analysis output

The Pandalyzer detected the mistake. The result of the analysis is `NOT OK`, and there is an error saying that all Dataframes ot be concatenated must have the same structure.

Case Study #4: High uncertainty, user input

The fourth case study again focuses on the uncertainty, but it shows different use cases of the uncertainty than the second case study.

This time, we do not have any configuration file. There is just a source code that is shown in the listing 5.13.

```
1  import pandas as pd
2
3  df = pd.DataFrame({
4      "string_column": [
5          input("First string: "),
6          input("Second string: "),
7          input("Third string: ")
8      ],
9      "int_column": [
10         int(input("First int: ")),
11         int(input("Second int: ")),
12         int(input("Third int: "))
13     ]
14 })
15
16 print(df[input("What column do you want to see? ")])
17
18 df["note"] = "User inserted the following string" \
19              + df["string_column"]
20
21 df.to_csv("output.csv")
```

Listing 5.13 Code for the fourth case study in Pandas

Even though the whole Dataframe `df` is constructed from the user input, the Pandalyzer should have enough information to derive its structure. The Pandalyzer should also be able to derive the type of the `note` column.

The listing 5.14 shows the actual output of the analysis.

```
Summary of analysis: OK
Global data structures (2):
pd: PandasImport
df: DataFrame(columns={string_column=StringType,
    int_column=IntType, note=StringType})

Warnings (11):
0: Assign from line 3 to line 14 columns 0 - 2:
    Unable to resolve result of input
1: Assign from line 3 to line 14 columns 0 - 2:
    Unable to resolve result of input
2: Assign from line 3 to line 14 columns 0 - 2:
    Unable to resolve result of input
3: Assign from line 3 to line 14 columns 0 - 2:
    Unable to determine the string value for int function
4: Assign from line 3 to line 14 columns 0 - 2:
    Unable to resolve result of input
```

```
5: Assign from line 3 to line 14 columns 0 - 2:
    Unable to determine the string value for int function
6: Assign from line 3 to line 14 columns 0 - 2:
    Unable to resolve result of input
7: Assign from line 3 to line 14 columns 0 - 2:
    Unable to determine the string value for int function
8: Assign from line 3 to line 14 columns 0 - 2:
    Unable to resolve result of input
9: ExpressionStatement on line 16 columns 0 - 52:
    The key for subscript of dataframe is not known
10: ExpressionStatement on line 16 columns 0 - 52:
    Unable to resolve result of input

Errors (0):

Output files (1):
File output.csv:
    string_column : StringType
    int_column : IntType
    note : StringType
```

Listing 5.14 Analysis output of the fourth case study

The first thing to notice is that the result of the analysis is `OK`. Another important information is that the Pandalyzer was actually able to derive the type of the `output.csv` file. The Pandalyzer knows that the `input` function returns string and the `int` function returns an int if it receives a string as an input. It does not need to know the exact values to construct the Dataframe structure. At last, there are eleven warnings. All of them are caused by the uncertainty and are mostly related to the `input` and `int` functions.

Case Study #5: Group by, aggregations

In the last case study, we go over some use cases of the group by operation and the associated aggregations. Let us have the following configuration file (shown in listing 5.15):

```
[all_strings.csv]
str1 = "string"
str2 = "string"
str3 = "string"

[first_string.csv]
col1 = "string"
col2 = "int"
col3 = "int"

[first_int.csv]
col1 = "int"
col2 = "string"
col3 = "string"

[all_different.csv]
int_col = "int"
str_col = "string"
bool_col = "bool"
```

Listing 5.15 config.toml file for the fifth case study

The source code for the case study can be seen in the listing 5.16.

```
1  import pandas as pd
2
3  all_strings_df = pd.read_csv("all_strings.csv")
4  first_string_df = pd.read_csv("first_string.csv")
5  first_int_df = pd.read_csv("first_int.csv")
6  all_different_df = pd.read_csv("all_different.csv")
7
8  pass_df1 = first_string_df.groupby("col1").mean()
9  fail_df1 = first_string_df.groupby("col2").mean()
10
11 pass_df2 = all_strings_df.groupby("str1").count()["str2"] + 3
12 fail_df2 = all_strings_df.groupby("str1").count()["str2"] + "hi"
13
14 pass_df3 = all_different_df.groupby("bool_col").sum()
15 fail_df3 = all_different_df.groupby("str_col").sum()
16
17 pass_df4 = first_int_df.groupby(["col2", "col3"]).mean()
18 fail_df4 = first_int_df.groupby("col3").mean()
```

Listing 5.16 Code of the fifth case study in Pandas

The code reads all four CSV files defined and stores them in the Dataframes. Then, there are four pairs group by operations followed by aggregations. Every pair has one operation that passes and one that fails.

The first pair applies the `mean` function on the `first_string_df` grouped by `col1` and by `col2`. However, only the first option passes. The reason is that the mean function is applied to all columns that we did not group by. So in the second case, it is also the `col1` column that is string. But the mean function cannot be applied on a string column.

The second pair shows the usage of `count` function on the `all_strings_df` grouped by `str1`. The count should return an int column for all columns that we did not group by. So summing some of the columns with a number should pass and summing it with string should fail.

In the third pair, we apply the sum function to non-grouped columns. In the first case, it is `str_col` and `int_col`, which can be summed up. In the second case, it is `int_col` and `bool_col`. The second case will actually **not** fail this time. But summing of bool column is usually something that we do not want, so we should consider it an error.

The last (fourth) case again computes a mean. This time, the first case will not fail, since the only column we apply mean on is `col1` which is an int column. The second case will fail, since we apply the mean on both the `col1` and `col2`, but the `col2` is a string column that we cannot apply mean on.

The listing 5.17 shows the output of the Pandalyzer on the code in the listing 5.16 and the configuration in the listing 5.15.

```
Summary of analysis: NOT OK
\* snip *\
Warnings (0):

Errors (4):
0: Assign on line 9 columns 0 - 49:
    Cannot apply mean on the columns: col1 of type StringType,
1: Assign on line 12 columns 0 - 67:
```

```
    Cannot sum a series of type IntType with PythonString
2: Assign on line 15 columns 0 - 52:
    Cannot apply sum on the columns: bool_col of type BoolType,
3: Assign on line 18 columns 0 - 46:
    Cannot apply mean on the columns: col2 of type StringType,

Output files (0):
```

Listing 5.17 Analysis output of the fifth case study

The analysis shows exactly what we said. There are four errors, each explaining one of the mistakes discussed.

Summary

The Pandalyzer is already capable of checking many various Pandas operations. It is also able to work with some degree of uncertainty as seen in the case studies. In each analysis, the Pandalyzer tells us the analysis result (`OK` or `NOT OK`), active variables in the scope, list of warnings, errors and all output files created together with their structure. The errors and warnings contain information about the line and column of the source statement, type of the statement and some message describing the issue. However, to be actually useful in practice, there is a need to implement other Pandas operations and Python constructs. The future of the Pandalyzer will be discussed in the Future Work 5.1 section of the Conclusion

Conclusion

The aim of this thesis was to design and implement a code-analysis tool for the Pandas library that would be capable of checking common kinds of errors such as access to misspelled or non-existent columns. The resulting system should have been evaluated through a set of realistic case studies. Let us recap and summarize to what extent we did that.

We first defined a framework for the Abstract Interpretation analysis of the Dataframe and Series type and then expanded it to other types such as strings, numbers, lists, etc. Then we used the framework to implement the Pandalyzer. The Pandalyzer is a code-analysis tool that uses Abstract Interpretation to interpret Python code with the Pandas library. The Pandalyzer is capable of detecting errors such as access to non-existent column, operation on incompatible types, incorrect function arguments, operations leading to an incorrect state. It also reports the structure of the output CSV files and is able to accept information about the structure of the input CSV files. The Pandalyzer is able to work with some amount of uncertainty generated from the user input. The supported functions include merge, groupby, drop, rename, read_csv, to_csv, concat, Dataframe creation, Series creation, the subscript operator in both get and set contexts and vectorized sums and products, aggregation function such as mean, sum, first, last, count or head. As shown in the case studies in the chapter 5, this set of functions already gives the user enough flexibility to do many various data manipulation tasks. Moreover, the Pandalyzer is highly extensible, so implementing support for other Pandas functions is possible.

On the other hand, the Pandalyzer has some limitations. It now only supports a subset of Python language, so the analyzer will not be able to proceed with the analysis when it encounters unknown Python construct. However, it is planned in the Future Work 5.1 to add support for the rest of the Python language.

The source-code of Pandalyzer can be found in the Pandalyzer GitHub repository [11], and the source-code for this thesis can be found in the bachelor-thesis GitHub repository [16].

Related Work

The idea to use Abstract Interpretation to analyze programs working with Dataframes was already proposed by Yungyu Zhuang and Ming-Yang Lu [7]. They also created a proof of concept implementation named PDChecker. However, PDChecker does not have as many checks, does not report output CSV files (via to_csv() function) and does not allow for interpreting multiple branches of if-statements and other sources of non-determinism.

Another related subject is the type hints [17] in Python language. The language itself does not enforce any typing rules, but we can put type annotations in the code, and analyzers integrated in the IDE can warn us in case that the types do not match. Unfortunately, this is not useful for our case, since the type annotations are only able to express that (for example) the function returns a Pandas Dataframe, not a Pandas Dataframe with a specific column structure.

The Types for Tables [18] article defines the Table API—the definition of a

table and operations on a table similarly as we did it in the chapter 3.

Future Work

There are a lot of plans ahead of us regarding the implementation. In the future, we would like to support more Pandas operations since Pandas is a large library with many useful operations. We would also like to add support for working with Indexes on Dataframes and Series. The Pandalyzer so far supports a limited subset of Python language constructs. We chose the most useful language constructs for data manipulation. However, the plan for the future is to add support for other Python constructs such as Lambdas, Match statements, For-loops or Classes. As of now, Pandalyzer is only able to analyze one module, so extending it to be able to analyze multiple modules together could be also a useful extension.

The Pandalyzer tool could be also extended for other related Python libraries. A good example is the numpy library for working with vectors, matrices, etc. The abstraction could be defined as the dimensions of the vectors and matrices. Another good example of library worth including to the Pandalyzer is the matplotlib library for data visualization. We could support reporting of what visualizations would be displayed to the user. This could be implemented in a similar way as the analysis of the to_csv function in Pandas.

Another field where the Pandalyzer could be extended is its integration to the IDEs such as PyCharm or VS Code. This could be done using the Language Server Protocol.

Bibliography

1. TEAM, The pandas development. *pandas-dev/pandas: Pandas*. Zenodo, 2020. Latest. Available from DOI: 10.5281/zenodo.3509134.
2. MÜLLER, Kirill; WICKHAM, Hadley. *tibble: Simple Data Frames*. 2023. https://tibble.tidyverse.org/, https://github.com/tidyverse/tibble.
3. R CORE TEAM. *base: Base R Functions*. 2019. Available also from: https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/data.frame. R package version 3.6.2.
4. JULIADATA. *DataFrames.jl: In-memory tabular data in Julia*. 2019. Available also from: https://dataframes.juliadata.org/stable/. Julia package version stable.
5. COUSOT, Patrick; COUSOT, Radhia. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, 1977, pp. 238–252. Available also from: https://www.di.ens.fr/~cousot/COUSOTpapers/POPL77.shtml.
6. BLANCHET, Bruno. Introduction to Abstract Interpretation. *Inria*. 2002. Available also from: https://bblanche.gitlabpages.inria.fr/absint.pdf.
7. ZHUANG, Yungyu; LU, Ming-Yang. Enabling Type Checking on Columns in Data Frame Libraries by Abstract Interpretation. *IEEE Access*. 2022, vol. 10, pp. 14418–14428. Available from DOI: 10.1109/ACCESS.2022.3146287.
8. JETBRAINS. *Kotlin Documentation*. 2024. Available also from: https://kotlinlang.org/docs/home.html. Accessed on April 28, 2024.
9. INC., Gradle. *Gradle User Manual*. 2024. Available also from: https://docs.gradle.org/current/userguide/userguide.html. Accessed on April 28, 2024.
10. LINDHOLM, Tim; YELLIN, Frank; BRACHA, Gilad; BUCKLEY, Alex. *Java Virtual Machine Technology Overview*. 2024. Available also from: https://docs.oracle.com/en/java/javase/22/vm/java-virtual-machine-technology-overview.html. Accessed on April 28, 2024.
11. HRUBIAN. *Pandalyzer* [https://github.com/Hrubian/Pandalyzer]. 2024.
12. PYTHON SOFTWARE FOUNDATION. *ast — Abstract Syntax Trees*. 2024. Available also from: https://docs.python.org/3/library/ast.html. Python 3.10.6 documentation.
13. KOTLINX SERIALIZATION CONTRIBUTORS. *Kotlin Serialization Guide*. 2024. Available also from: https://github.com/Kotlin/kotlinx.serialization/blob/master/docs/serialization-guide.md.
14. PEUCH, Laurent. *ast2json Python Module*. 2024. Available also from: https://github.com/YoloSwagTeam/ast2json.

15. Preston-Werner, Tom. *TOML: Tom's Obvious Minimal Language*. 2021. Tech. rep. Available also from: https://toml.io/en/.
16. Hrubian. *bachelor-thesis* [https://github.com/Hrubian/bachelor-thesis]. 2024.
17. Python Software Foundation. *typing — Support for type hints* [Python 3.12.3 documentation]. 2023. Available also from: https://docs.python.org/3/library/typing.html.
18. Lu, Kuang-Chen; Greenman, Ben; Krishnamurthi, Shriram. Types for Tables: A Language Design Benchmark. *CoRR*. 2021, vol. abs/2111.10412. Available from arXiv: 2111.10412.

List of Figures

2.1 Schema of a Series 18
2.2 Schema of a Dataframe 20

3.1 Abstract structures hierarchy 29
3.2 Nondeterministic structure 30

4.1 AST representation hierarchy 34

Listings

1 Pandas code with errors 7
2.1 Working with Series in Pandas 17
2.2 Index on a Series 18
2.3 Heterogenous Series 18
2.4 Working with Dataframe in Pandas 19
2.5 Index on a Dataframe 19
2.6 Select in Pandas 20
2.7 Where in Pandas 21
2.8 As in Pandas 21
2.9 Order by in Pandas 21
2.10 Join in Pandas 22
2.11 Group by in Pandas 22
2.12 Having in Pandas 23
2.13 Vectorized operations 23
4.1 Kotlin definition of the Dataframe 36
4.2 Kotlin definition of the Series 36
4.3 Snippet of attribute function of Dataframe 37
4.4 The rename function of DataFrame_RenameFunc 37
4.5 An example configuration file 40
4.6 An example regex-based file definition 40
5.1 config.toml of the first case study 42
5.2 Solution of the first case study in Pandas 43
5.3 Output of Pandalyzer on the first case study 43
5.4 Misspelled column on rename function and analysis output 44
5.5 Incorrectly specified argument and analysis output 45
5.6 Code of the second case study in Pandas 45
5.7 config.toml file for the second case study 46
5.8 Analysis output of the second case study 46
5.9 config.toml of the second case study 47
5.10 Solution of the third case study in Pandas 47
5.11 Analysis output of the third case study 48
5.12 Incompatible Dataframes to concat operation and analysis output 48
5.13 Code for the fourth case study in Pandas 49
5.14 Analysis output of the fourth case study 49
5.15 config.toml file for the fifth case study 50
5.16 Code of the fifth case study in Pandas 51
5.17 Analysis output of the fifth case study 51

List of Abbreviations

Poset - Partially ordered set
AST - Abstract syntax tree
IDE - Integrated development environment
CSV - Column separated values
pc - program counter

A User documentation

A.1 Building from source

To build the Pandalyzer from sources, follow the steps below:

1. Ensure that you have Java (version 21.0.1 or higher), Git and Python 3.x installed.
2. Clone the Pandalyzer repository:

   `git clone https://github.com/Hrubian/Pandalyzer.git`
3. Navigate to the root folder of the repository:

   `cd Pandalyzer`
4. Run the Gradle bootstrap script:

   `./gradlew build` or `./gradlew.bat build` on Windows

A.2 Running the tool

The build generates a `./build/` folder. Check that there are also `./build/distributions/Pandalyzer.tar` and `./build/distributions/Pandalyzer.zip` archives. Unpack one of them (depending on what tools you are provided with) and run the `Pandalyzer` (or `Pandalyzer.bat`) script in the `bin` folder. The program accepts the following command-line arguments:

- `-h, --help` - Prints usage information and exits
- `-i, --input <arg>` - The input python script to analyze **(mandatory)**
- `-o, --output <arg>` - The output file to store the analysis result to (standard output by default)
- `-c, --config <arg>` - The configuration file to read the file structures from (config.toml by default)
- `-f, --format <arg>` - The format of the analysis output, possible options: hr (human-readable), json (hr by default), csv