Automated Checks of Implicit Assumptions on Textual Data

Radwa Sherif Abdelbar
Supervisors: Dr. Caterina Urban, Alexandra Bugariu

March 2018

1 Introduction

In the field of data science, tremendous effort and time are invested in data preparation and data cleaning. A recent survey [1] among data science professionals shows them to list "dirty data" among the most challenging problems of the field. [2] presents a taxonomy of dirty data which classifies it into three main categories: missing data, wrong data and unusable data.

Missing data includes null values where a no-null-allowed constraint is supposed to be enforced. Wrong data can occur in the form of wrong data types, out-of-range values and duplicate data that violates uniqueness constraints. Unusable data is neither missing nor wrong, but might yield incorrect results for an analysis or query. For example, an inconsistency between the age and birth date fields, including different values for the salary of an employee or different representations of date values. Unusable data is usually a result of data maintained across different databases or non-standard representation of some values (e.g. dates).

If programmers are not careful in the assumptions they make about the input data (for example, assumptions about the absence of null values or specific data formats) the program might crash. More dramatically, the program might continue to run normally, but produce incorrect results. This is especially hard to detect in the case of data science algorithms, where the influence of wrong input data on the output of the algorithm cannot be quantified due to the statistical nature of such algorithms.

In the next section, we present two code examples in which programmer assumptions about input data might cause various problems.

2 Example

Listings 1 and 2 are excerpts from a program that performs genetic ancestry analysis in Python.

In Listing 1, the programs reads input from the two files at the same time. It extracts the reference and alternate base from the file *yri.csv* and uses this information to translate the DNA representation in *anon.csv* from using letters (A and G) to using numbers (0 and 1).

```
1  yri = open('yri.csv')
2  anon = open('anon.csv')
3
4  for line in yri:
5      row = line.split(",")
6      g = anon.next().strip().split(",")[4]
7      ref = row[3]
8      alt = row[4]
9
10     if g == ref+ref:
11         print("0|0")
12     elif (g == ref+alt) | (g == alt+ref):
13         print("0|1")
14     elif g == alt+alt:
15         print("1|1")
16     else:
17         raise ValueError
```

Listing 1: Example of Assumptions on String Structure

From Lines 5 and 6, one can infer the assumption that the two lines contain comma-separated rows, one data row per input file line. Lines 6 through 8 make assumptions about the number of elements present in each row. For example, if one line in the *yri.csv* file has less than five elements, an error will occur on Line 8. The if-else statement starting on Line 10 makes the assumption that the variable **g** is one of four combinations resulting from the concatenation of the variables **alt** and **ref**. If that is not the case, a ValueError is thrown.

Listing 2 attempts to extract a subset of the information out of the file *yri.csv*. Again, we have the assumption of the comma-separated values. Lines 4 and 5, however, make a more complex assumption. They assume that we have at least ten values in the array and that all elements starting from index 9 are separated by a colon.

```
1  in_file = open('yri.csv')
2  for line in in_file:
3      row = line.strip().split(",") #read from yri
4      genotype = row[9:]
5      genotype = [i.split(":")[0] for i in genotype]
6      print genotype
```

Listing 2: Example on More Complex Data Properties

We have observed, by inspection of code examples, several interesting types of assumptions about textual data that are present in Python code. These types are related to the properties listed below:

- The layout of the data in the input file. For example, a program could assume that its input is a list of strings distributed one per line in the input data file while in reality the strings are distributed in a more arbitrary fashion. (Listing 1, lines 5 and 6).

- The composition of the textual data. This includes assumptions about the alphabet composing the text, for example that it consists only of lower case English alphabets, and assumptions about the structure of the text such as that it matches a certain regular expression (Listing 2, line 5).
- Dictionary key assumptions, such as taking for granted the presence of a certain set of strings as keys in a dictionary while in may be the fact that those strings are missing.
- Relational assumptions between strings. For example, assuming that a string is a concatenation of several others (Listing 1, lines 10 through 17).

3 Related Work

A variety of techniques and tools have been developed for the purpose of data cleaning. [3] lists some data cleaning approaches and some of the tools which implement them. Data profiling tools, for instance, are specialized in collecting metadata about each attribute of the input data, this metadata is then used to detect errors in the data. On the other hand, data mining tools are concerned with inferring relationships between different data fields and checking integrity constraints among attributes. Another category of tools are domain specific, such as tools that focus on validating and formatting names and addresses, and others that specialize in duplicate elimination.

Another input checking tool is CheckCell [4], an add-in for Microsoft Excel and Google Spreadsheets, which presents an approach called data debugging. CheckCell works by performing statistical analysis on the input data and pointing out input cells that have a disproportionately large impact on the output, the underlying assumption being that the value of the output changes significantly when an erroneous data value is corrected.

4 Contribution

All of the aforementioned approaches are characterized by the fact that they require the input data itself to be available in order to validate its correctness. In fields such as medicine and genetics, data may not be available due to privacy issues and regulations.

In this thesis, we present an approach to ensure the correctness of input data that functions independently of the data itself. Our approach uses static analysis of programs to infer implicit assumptions made about the input data such that if the data does not fulfill those assumptions the program crashes or produces incorrect results. Then, these assumptions are fed into an input checker, integrated in a text editor, which highlights to the user the input values which do not fulfill the assumptions and are likely to cause failure or erroneous output.

We build on the progress achieved in [5]. We extend the static analyzer which currently infers assumptions on numerical values to also infer assumptions about string values and textual data. It infers non-relational assumptions about the type and possible ranges of numerical data, while supporting only a limited form of relational assumptions. We will extend the analyzer to provide full support for relational assumptions that can be represented using the octagon domain [6]. The input checker will also be extended to accommodate for the new features added to the static analyzer and will be implemented as a text editor plugin for a better user experience.

5 Core goals

- *Collecting examples.* We collect Python code samples from various sources with the aim of finding interesting assumptions to account for in our static analysis. Sources of code samples include Codeforces, an online competitive programming platform which makes accepted solutions code available for inspection. Online courses related to genomic data processing are also an interesting source for code samples, such as that in Listing 2, due to the variety of string processing operations carried out by those programs.

- *Analysis design.* Our analysis will be based on the Abstract Interpretation framework [7], which aims to approximate program behaviour with regard to certain properties using computer-representable objects. A design decision will be made about which of the string abstract domains available in the literature [8, 9] should be used, combined or adapted to suit our problem. Building on the work of the previous thesis, we will implement an over-approximation of the program semantics. We will infer the preconditions necessary to make a program run correctly. Section 3 as well as Listings 1, 2 and 3 refer to some of the assumption types we are interested in.

- *Support for octagon domain.* Numerical analysis is indispensable for string analysis. For example, we need to infer assumptions about string lengths and indexes. The current analyzer supports relational assumptions on numerical data by implementing only a subset of the octagon domain. It supports order, join, meet and widening operators without a closure algorithm and does not support the filter operator. The filtering operator is limited to relations of the form x $\bullet$ y where $\bullet \in \{>, <, \geq, \leq\}$. The relational assumption on string lengths on Line 3 in Listing 3 would be missed under the current analysis. We will extend the analyzer to support relations that can be represented by octagon domains using the Python interface of the Elina library [10].

- *Input checker implementation.* The input checker will be enhanced to check for assumptions inferred by the new static analyzer. The current input checker is implemented as a standalone tool with a graphical user

interface. A user study conducted in the previous thesis showed that the tool would provide a better user experience if implemented as a plugin to a text editor or an IDE. We will also implement this.

- *Evaluation.* The analysis and input checker will be evaluated using two methods: comparison of assumptions produced by the analysis with the assumptions produced by manual inspection of code examples and a user study to test the usability of the input checker.

```
CHARS_PER_LINE = 100
s1, s2 = raw_input(), raw_input()
if len(s1) + len(s2) < CHARS_PER_LINE:
    print s1,s2
else:
    raise ValueError
```

Listing 3: Relational assumption on string lengths

6 Extension goals

- *Analysis with under-approximation.* [5] performs an over-approximation of the program semantics, meaning that it does not allow for false positives. It presents the user with the necessary errors to fix so that the program may function correctly. As an extension goal, we would like to implement an analyzer that uses under-approximation, meaning that it may wrongly flag correct input data as erroneous, but in return guarantees that no errors are missed. We would like to then combine the two approaches to achieve as precise an approximation as possible.

- *Locating missing input lines.* So far the input checker provides no information about the location of missing data lines, the assumption being that they are located at the end of the file. We would like to implement an algorithm that detects the possible location of missing data lines.

- *Synthesis of data cleaning code.* So far the input-checker lets the user correct the problematic input values manually. We would like to generate code that performs data cleaning for the user automatically.

- *Support for function calls.* The analyzer currently supports sequential code execution inside one procedure. We would like to extend the analysis to be inter-procedural.

References

[1] Kaggle: The state of data science and machine learning, 2017.

[2] Won Kim, Byoung-Ju Choi, Eui-Kyeong Hong, Soo-Kyung Kim, and Doheon Lee. A taxonomy of dirty data. *Data Mining and Knowledge Discovery*, 7(1):81–99, Jan 2003.

[3] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[4] Daniel W Barowy, Dimitar Gochev, and Emery D Berger. Checkcell: data debugging for spreadsheets. In *ACM SIGPLAN Notices*, volume 49, pages 507–523. ACM, 2014.

[5] Madelin Schumacher. Automated generation of data quality checks. Master's thesis, ETH Zurich, 2018.

[6] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1):31–100, 2006.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[8] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static analysis of string values. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, pages 505–521, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. Combining string abstract domains for javascript analysis: An evaluation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–57, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[10] Gagandeep Singh, Markus Püschel, and Martin Vechev. A practical construction for decomposing numerical abstract domains. *Proceedings of the ACM on Programming Languages*, 2(POPL):55, 2017.