Automated Checking of Implicit Assumptions on Textual Data

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

August 13, 2018

Abstract

1 Introduction

Today, we live in a world that produces tremendous amounts of data on a daily basis. The extensive use of social media websites and the digitization of traditional services such as banking, retail and publishing guarantees the existence of large datasets that are available to service providers. In addition to business, technological advancements have created an abundance of data in many fields of scientific research such as the genomic data created by efficient DNA sequencing or raw images of celestial bodies that are captured by modern telescopes [1].

This abundance of data, along with the rapid development in new data science techniques and methods to organize, analyze and detect patterns in large datasets, creates new and unique opportunities for experts in different domains. For example, it enables businesses to predict future customer behavior or medical researcher to associate the presence of a specific gene in the human DNA with susceptibility to certain diseases.

These advantages, however, are not achieved without challenges or difficulties. The datasets processed by data science algorithms are typically very large in size and could be obtained from various sources or maintained on different machines. For this reason, they are likely to contain errors and inconsistencies.

In the field of data science, raw data, that is data as it is collected from the source and stored on some storage medium (e.g. database or cloud), is not usually usable as is. A recent survey among data scientists [2] shows them to cite "dirty data" as one of the most challenging aspects of their work. Dirty data, as defined in [3], is data that is wrong, missing or not represented according to a known standard such as non-standard representation of time and date.

Before feeding the data into a data science program, it must be cleansed and the erroneous values must be eliminated. Many tools have been introduced to achieve this goal.. [mention some data cleaning tools.]

In this thesis, we present an approach to detect incorrect input data to Python programs. Unlike the data-cleaning tools mentioned above, our method does not perform computations on the dataset directly. Instead, we analyze the source code of the program which is to run on this data and try to compute the set of input values that will allow the program to terminate without raising any errors. We then scan the dataset line by line to see if any existing values are outside this set of accepted values. If that is the case, the wrong input values are flagged as errors so that the user can correct them.

```
1  sequence_length: int = int(input())
2  count_a: int = 0
3  count_c: int = 0
4  count_g: int = 0
5  count_t: int = 0
6  for i in range(sequence_length):
7    base: str = input()
8    if base == 'A':
9      count_a: int = count_a + 1
10   elif base == 'C':
11     count_c: int = count_c + 1
12   elif base == 'G':
13     count_g: int = count_g + 1
14   elif base == 'T':
15     count_t: int = count_t + 1
16   else:
17     raise ValueError
```

Listing 1.1: Example of Assumptions on String Data

We compute the set of accepted input values to a program by means of a static analysis that detects assumptions made by a program about its input data. To elaborate further, let us examine the program in Listing 1. This program aims to read a DNA sequence and count the frequency of occurrence of each nucleotide. It first reads the length of the sequence, then since the only possible nucleotides in a DNA sequence are represented by the letters A, C, G and T, it sets a frequency counter for each of these nucleotides. Then the program iterates through the sequence, reading one letter per line and incrementing the corresponding counter variable. If it encounters a letter that is neither A, C, G or T, it raises an exception because it *assumes* that, in an ideal scenario, no value in the input file will be outside this set of letters.

Our aim, thus, is not to perform computations on the input data based on the known fact that the human DNA contains only A, C, G and T bases and to cleanse the wrong values. Rather, we turn our attention to the program that will run on the data and try to answer the question: what does this program *assume* about its input data such that if those assumptions are not fulfilled, the program will raise an error?

Our approach is composed of two main steps. First, we use static analysis to infer assumptions that the program made about the input data. Then, using the output of the first step, we run an input-checking algorithm on the data file in order to find the errors.

[explain outline of the thesis]

1.1 Theoretical Background

1.1.1 Static Analysis

[Explain what static analysis is on a high level, what it's used for, the ways to do it]

1.1.2 Abstract Interpretation

[Explain briefly abstract interpretation and the advantages of using it in static analysis, element of an abstract domain.]

1.2 Previous Work

We build our approach on the work done in [4] which focuses on inferring assumption on numerical data using the Interval Domain and a specially designed relations domain. Our main contribution is designing a generic framework that can use any existing abstract domain to keep track of assumptions on the input data.

2 Static Analyis

2.1 Concrete Domain

To define our concrete domain, we first define the following sets:

- $\mathcal{L}$: the set of all program points in the program being analyzed.
- $\mathcal{S}$: the set of all possible strings.
- $\mathcal{V}$: the set of program variables.
- Following from the previous definitions, $\wp(\mathcal{V} \to \mathcal{S})$ is a set of mappings from program variables to the possible values they can take. Note that one program variables is allowed to be mapped to multiple values.
- We also take $\wp(\mathcal{S})^*$ as the list of values read as input from one program point onwards.

We can then define our concrete domain as follows:

$$\mathcal{L} \to \wp(\mathcal{V} \to \mathcal{S}) \times \wp(\mathcal{S})^*$$

As indicated by the formula, our concrete domain maps a program point to a set of mappings of program variables to the values they are allowed to take and a list of input values which the program reads from this point onwards such that the program terminates without any errors.

2.2 The Assumption Abstract Domain

We use the theory of Abstract Interpretation [5] to design an abstract domain that over-approximates the concrete semantics defined in the previous section. In this context, over-approximation means that the constraints inferred by the analysis are necessary but not sufficient for the program to run without producing an error. If the input values violate those constraints, the program is guaranteed to produce an error. We guarantee an absence of false-positives. However, if the input values satisfy all the constraints, the program might still produce an error. Our static analysis works backwards.

To examine the properties of our domain, let us consider the program in Listing 2, which performs the same operation as Listing 1, namely reading a DNA sequence and counting the frequency of every nucleotide, but on multiple sequences separated by a '.' or '#' character. On line 1, the number of sequences is read. Lines 3 and 4 assert that we have at least one sequence in the file. Another addition in Listing 2 is the check that the sequence length is not greater than some maximum length on lines 7 and 8.

There are multiple ways in which this program can produce an error. On line 1 if the value read cannot be cast to an integer, a error will be raised by Python. If the number of sequences is not positive (line 3), the length of each sequence is greater than the designated maximum sequence length, the sequence contains characters other than A, C, G and T (lines 15 through 24) or if the separator character is not a hash or a dot (line 26), the program raises a ValueError explicitly.

It is obvious from this example that we need to track information about both numerical values and string values if we are to compute the set of allowed input values to this program. For examples we need to ensure that the relation $sequence\_length > max\_length$ holds and that the variable *base* has only the characters in the set $\{'A','C','G','T'\}$. This cannot be achieved by running a conventional static analysis using only one abstract domain.

```
1  number_of_sequences: int = int(input())
2  max_length: int = int(input())
3  if number_of_sequences <= 0 :
4    raise ValueError("Expecting at least one DNA sequence
     ")
5  for s in range(number_of_sequences):
6    sequence_length: int = int(input())
7    if sequence_length > max_length:
8      raise ValueError
9    A_count: int = 0
10   C_count: int = 0
11   G_count: int = 0
12   T_count: int = 0
13   for i in range(sequence_length):
14     base: str = input()
15     if base == 'A':
16       A_count: int = A_count + 1
17     elif base == 'C':
18       C_count: int = C_count + 1
19     elif base == 'G':
20       G_count: int = G_count + 1
21     elif base == 'T':
22       T_count: int = T_count + 1
23     else:
24       raise ValueError
```

```
25      separator: str = input()
26      if separator == '.' or separator == '#':
27        pass
28      else:
29        raise ValueError
```

Listing 2.1: Example of Assumptions on both String and Numerical Data

Our abstract domain, which we call the Assumption Domain, is a generic domain that allows for the approximation of multiple program properties simultaneously. It is parametrized by a list of abstract domains which are independent of one another. For each step of the analysis, the Assumption Domain invokes the corresponding operator or transformation on each domain independently. Thus, in the case of the example above, we can use the Octagons Domain [6] to keep track of the relations between numerical variables such as $sequence\_length \leq max\_length$ and the Character Inclusion Domain [7] to keep track of the characters of the variable *base*.

Another important property of our abstract domain is its ability to store assumptions about inputs read by the program in any of its scopes. Going back to Listing 2, let us examine the variable *base*. If we trace a backward static analysis from line 24 to line 15, we will get some information about this variable. For example, using the Character Inclusion Domain, we will get that this variable is allowed to contain only the characters 'A', 'C', 'G' and 'T'. However, on line 14 whatever information we have captured before will be lost with the assignment statement `base = input()`. Therefore, we need a data structure in which to store the constraints computed so far about this variable. At the end of the analysis, this data structure will contain the constraints about all the inputs read in course of the execution of the program.

In our case, we choose this data structure to be a stack, in which every layer represents a scope of the program. Whenever the analysis enters a new scope of the program, a new layer is pushed onto the stack. Inside of this scope, whenever a value is read as input, we store its constraints on the top layer of the stack. When existing a scope, the top layer is popped form the stack. This way, at the end of the analysis, we have a stack with one layer containing all the assumptions about the inputs read in all the scopes of the program.

Given the previously mentioned properties, we introduce our domain formally below.

We define $SUBD$ to be a family of abstract domains which are suitable for our analysis. In section 2.2.1, we explain the properties of these domains in detail.

In addition, we define $STACK$ to be a set of all possible stacks which store assumptions on input values of the program. The stack data structure and its operations are defined in detail in section 2.2.2.

We use the notation $(X_i)_{i=1}^n$ throughout this thesis to indicate a list of length n.

We define the Assumption Domain formally as follows:

$$D \equiv SUBD^n \times STACK$$

- An element $d \in D = \{((S_i)_{i=1}^n, Q) | S_i \in SUBD \wedge Q \in STACK\}$. Every element of this domain consists of a sequence of instances of sub-domains and a stack. Every instance of a sub-domain S_i keeps track of all the variables at the same time. If a variable does not belong to a type the domain can handle, it is always mapped to top in that domain. For example, if a variables is of type string and the domain S_1 is the Interval Domain, then x will always be mapped to top in S_1.

- A concretization function $\gamma_D(d) = (\bigcap_{i=1}^{n} \gamma_{S_i}(S_i), \gamma_{STACK}(Q))$. From the previous definition follows the concretization function. It is the intersection of concretization of all the sub-domains as well as the concretization of the stack.

- A partial order $\sqsubseteq_D$ such that $((S_{1,i})_{i=1}^n, Q_1) \sqsubseteq_D ((S_{2,i})_{i=1}^n, Q_2) \iff \bigwedge_{i=1}^{n} (S_{1i} \sqsubseteq_{S_i} S_{2i}) \wedge Q_1 \sqsubseteq_{STACK} Q_2$. An element d_1 of D is less than or equal to another element d_1 if and only if every instance of a sub-domain in d_1 is less than or equal to the corresponding instance of the same sub-domain in d_2 and if and only if the stack of d_1 is less than or equal to the stack of d_2.

- A minimum element $\bot_D = ((\bot_{S_i})_{i=1}^n, \bot_{STACK})$.

- A maximum element $\top_D = ((\top_{S_i})_{i=1}^n, \top_{STACK})$

- A join operator $\sqcup_D$ such that $((S_{1,i})_{i=1}^n, Q_1) \sqcup_D ((S_{2,i})_{i=1}^n, Q_2) = (((S_{1,i} \sqcup_{S_i} S_{2,i})_{i=1}^n, Q_1 \sqcup_{STACK} Q_2))$. The join operator is applied pair-wise to all sub-domains and the stack. Every instance of a sub-domain in the first element is joined with the corresponding instance of the same domain in the second element. The two stack are also joined.

- A meet operator $\sqcap_D$ such that $((S_{1,i})_{i=1}^n, Q_1) \sqcap_D ((S_{2,i})_{i=1}^n, Q_2) = (((S_{1,i} \sqcap_{S_i} S_{2,i})_{i=1}^n, Q_1 \sqcap_{STACK} Q_2))$. Similar to the join, the meet operator is applied pair-wise to all sub-domains and the stack.

- A backward assignment operator $[\![X := aexpr]\!]((S_i)_{i=1}^n, Q) = (([\![X := aexpr]\!](S_i))_{i=1}^n, [\![X := expr]\!](Q))$. The backward assignment operator is applied element-wise to every sub-domain and to the stack.

- A filter operator $[\![bexpr]\!]((S_i)_{i=1}^n, Q) = (([\![bexpr]\!](S_i))_{i=1}^n, [\![bexpr]\!](Q))$. The filter operator is applied element-wise to every sub-domain and to the stack.

- A widening operator ∇_D such that $((S_{1,i})_{i=1}^n, Q_1)\nabla_D((S_{2,i})_{i=1}^n, Q_2) = ((S_{1,i}\nabla_{S_i} S_{2,i}), Q_1 \nabla_{STACK} Q_2)$. Similar to the join and meet, widening is applied pair-wise between the sub-domains and the stack.

2.2.1 Sub-domains

As mentioned in the previous section, our Assumption Domain can make use of any existing abstract domain in order to track assumptions on input data. However, in order for this setting to work effectively, we need to define some extra operators for existing abstract domains.

$SUBD$ is the family of abstract domains which are capable of keeping track of constraints on variables. The variables are either program variables or special variables of the form $l[program\_point]$ that represent a program point. An element $F \in SUBD$ is an abstract domain whose concretization function γ_F operators $\sqcup_F, \sqcap_F, \sqsubset_F, \nabla_F$, backward assignment and filter are already defined.

The domains used in our analysis are required to define a special operator $\mathcal{R}_F$ such that $\mathcal{R}_F(f, v, l)$ for $f \in F, v \in \mathcal{V}, l \in \mathcal{L}$ replaces every occurrence of a variable v with the program point l from which this variable is read as input.

In Listing 2, if we have an element in some relational domain F that keeps track of the constraint $sequence\_length \leq max\_length$ on line 7, then on line 6, we will call $\mathcal{R}_F(\{sequence\_length \leq max\_length\}, sequence\_length, l6)$ and then we get the element $\{l6 \leq max\_length\}$. Again on line 2, we call $\mathcal{R}_F(\{l6 \leq max\_length\}, max\_length, l2) = \{l6 \leq l2\}$. We, thus, obtain a constraint that is no longer between program variables, but between program points from which these variables are read as inputs.

```
1  x: int = input()
2  if x > 10:
3    y: int = int(input())
4    if y + x <= 10:
5      raise ValueError
6  else:
7    z: float = float(input())
8    if z + x <= 20:
9      raise ValueError
10
```

Listing 2.2: Example of unification

We also the sub-domains to define a special operator $\mathcal{U}(f_1, f_2)$ that unifies the environment of two elements of a relational domain. To illustrate what we mean by unification we introduce Listing 3. In the then-branch of the if-statement, we read a variable y and assert the condition $y + x > 10$. In the else-branch, we read a variable z and enforce the condition $z + x > 20$. Assuming that we have a relational domain that is capable of keeping track of these relations in the two branches. If we trace a backward analysis on this program and employing the $\mathcal{R}_F$ we defined earlier in this section, then in the then branch, we will get the

relation $l3{+}x > 10$ and in the else branch, we will get the relation $l7{+}x > 20$. A join needs to be performed on these two elements at the head of the if-statement. Since our analysis treats $l3$ and $l7$ as different variables, the join will yield top.

It is possible, however, to do better than this. From the perspective of our analysis, we do not care about the particular variable that represents an input value, but rather about the order in which inputs are read. Therefore in the case of Listing 2.2, we care that after reading x on line 1, we will read one more value, regardless of which branch the program will take. The function of the unification operator then is to enable us to treat both $l3$ and $l7$ as the same variable since, essentially, they reflect the same order of reading inputs in the program. For example, $\mathcal{U}_F$ can be defined to replace the later program point with the earlier one as follows: $\mathcal{U}_F(\{l3{+}x > 10\}, \{l7{+}x > 20\}) = \{l3{+}x > 20\}$. Then a join can be performed between $\{l3 + x > 10\}$ and $\{l3 + x > 20\}$.

2.2.2 The Stack

The stack used in our analysis follows the intuitive definition of a stack. It is composed of layers and defines push and pop operations. To introduce our stack, we need to define a set $B = \{(l, (c_i)_{i=1}^n) \mid l \in \mathcal{L} \wedge \exists_{F_{in} \in SUBD_{in}} c \in F_{in}\} \cup \{\star\}$ which is either a sequence of constraints from any domain in $SUBD$ associated with a specific program point or a constraint represented by $\star$ which is an empty constraint and is used to indicate a lack of information on what constraints are placed on input values. For example, in Listing 2, an element $(l7, (int, l7 \leq l2)) \in B$ would indicate that the value read from program point 7 is of type integer and is less than the value read form program point 2.

We then define the set of stack layers $I = \{m \times (a_i)_{i=1}^k \mid m \in \mathbb{M} \wedge a_i \in I \cup B\}$ as a set of possibly repeated constraints on the input data, where $\mathbb{M}$ is a set of multipliers that indicate how many time the constraints in the list are repeated. A multiplier is either an expression or an integer. For clarity, we express a list of constraints of length k that is repeated m times using the notation $m \times (a_i)_{i=1}^k$. We define a concretization function as well as join, meet and widening operators for the stack layers before we proceed to define them for the whole stack.

- A concretization function γ_I is defined recursively as follows:
  - $\gamma_I(\star) = \mathcal{S}$. An empty constraint indicates that the input value can be anything.
  - For $(l, (c_i)_{i=1}^n) \in B$, $\gamma_I((l, (c_i)_{i=1}^n)) = \gamma(c_1) \cap ... \cap \gamma(c_n)$. A tuple of constraints associated with one program point concertizes to any value that satisfies all of the constraints of this tuple.
  - For $m \times (a_i)_{i=1}^k \in I$, $\gamma_I(m \times (a_i)_{i=1}^k) = [\gamma_I(a_1), .., \gamma_I(a_k)]^m$. To concertize a constraint in the set I, the concretization function is applied recursively to its list of constraints, then the result is repeated as many times as the value of its multiplier.
- A partial order $\sqsubseteq_I$:

- For any $b \in B,\ b \sqsubseteq_I \star$.
- For $(l_1, (c_{1,i})_{i=1}^n), (l_2, (c_{,2i})_{i=1}^n) \in B$, the partial order is given by $\bigwedge_{i=1}^{n} c_{1,i} \sqsubseteq_I c_{2,i}$. We take the conjunction of the pair-wise order of the elements of the two tuples.
- For $m \times (a_{1,i})_{i=1}^k, m \times (a_{2,i})_{i=1}^k \in I$, that is, two elements in I with the same multiplier and the same number of constraints, the order is given by $\bigwedge_{i=1}^{k} a_{1,i} \sqsubseteq_I a_{2,i}$.
- For two elements where one belongs to I and the other to B, we default to false.

- A maximum element $\top_I \equiv 1 \times [\star]$.
- A minimum element $\bot_I$.
- A join operator $\sqcup_I$:
  - For $b \in B,\ b \sqcup_I \star = \star$
  - For $(l_1, (c_{1,i})_{i=1}^n), (l_2, (c_{2,i})_{i=1}^n) \in B$, the join is given by $(min(l_1, l_2), (c_{1,i} \sqcup c_{2,i})_{i=1}^n)$. The soundness proof of this join follows from the soundness of the join operator of the individual domains as follows: $\gamma_I((l_1, (c_{1i})_{i=1}^n)) \cup \gamma_I((l_2, (c_{2i})_{i=1}^n)) = (\gamma_I(c_{1,1}) \cap ... \cap \gamma_I(c_{1,n})) \cup (\gamma_I(c_{2,1}) \cap ... \cap \gamma_I(c_{2,n})) \subseteq \gamma_I(c_{1,1} \sqcup c_{2,1}) \cap ... \cap \gamma_I(c_{1,n} \sqcup c_{2,n})$.
  - For $m \times (a_{1,i})_{i=1}^k, m \times (a_{2,i})_{i=1}^k \in I$, that is, two elements in I with the same multiplier and the same number of constraints, the join is given by $m \times (a_{1,i} \sqcup a_{2,i})_{i=1}^k$. The soundness can be proved as follows: $[\gamma_I(a_{1,1}), ..., \gamma_I(a_{1,k})]^m \cup [\gamma_I(a_{2,1}), ..., \gamma_I(a_{2,k})]^m \subseteq [\gamma_I(a_{1,1} \sqcup a_{2,1}), ..., \gamma_I(a_{1,k} \sqcup a_{2,k})]^m$.
  - For $1 \times (a_{1,i})_{i=1}^{k_1}, 1 \times (a_{2,i})_{i=1}^{k_2} \in I$, where $k_1 \neq k_2$, the join is given by $m \times (a_{1,i} \sqcup a_{2,i})_{i=1}^{min(k1,k2)} \oplus [\star]$. When joining to elements from I with different constraint lengths, it is inevitable that we lose some information. The addition of the $\star$ constraint is an indication that there is at least one input value for which we have no constraints and which can take any possible value. The soundness proof is somewhat similar to the previous point. $[\gamma_I(a_{1,1}), ..., \gamma_I(a_{1,k})] \cup [\gamma_I(a_{2,1}), ..., \gamma_I(a_{2,k})] \subseteq [\gamma_I(a_{1,1} \sqcup a_{2,1}), ..., \gamma_I(a_{1,k} \sqcup a_{2,k}), \mathcal{S}]$.
- A meet operator returns the first element since it is not needed by the analysis for the set I.
- A backward assignment operator $[\![X := aexpr]\!]$ that is applied individually to constraints belonging to the set B.
- A widening operator $\nabla_I \equiv \sqcup_I$.

- A special replacement operator $\mathcal{R}_I$ that is similar to the $\mathcal{R}_F$ operator defined in the previous section:
  - For $(l, (c_i)_{i=1}^n) \in B$, the replacement operator works as follows:
    $$\mathcal{R}_I((l, (c_i)_{i=1}^n), v, l_1) = (l, (\mathcal{R}_F(c_i, v, l_1))_{i=1}^n)$$
    The respective operator of each domain is applied to the constraint belonging to that domain.
  - For elements of the set I:
    $$\mathcal{R}_I(m \times (a_i)_{i=1}^k) = m \times (\mathcal{R}_I(a_i)_{i=1}^k)$$
    The replacement operator is applied recursively to the constraints until it reaches an element of B, then it applies the $\mathcal{R}_F$ operator.
- An insertion operator $\mathcal{I}_I$ that is responsible for inserting new constraints or updating existing constraints on a stack layer:
  - For two elements $(l, (c_{1,i})_{i=1}^n), (l, (c_{2,i})_{i=1}^n) \in B$ that are associated with the same program point, we update the existing constraint by joining the two: $\mathcal{I}_I((l, (c_{1,i})_{i=1}^n), (l, (c_{2,i})_{i=1}^n)) = ((l, (c_{1,i})_{i=1}^n) \sqcup_I (l, (c_{2,i})_{i=1}^n))$.
  - For two elements $m \times (a_{1,i})_{i=1}^{k1}, m \times (a_{2,i})_{i=1}^{k2} \in I$ where $k1 \leq k2$ then it is required to update the existing constraints as follows: $m \times (\mathcal{I}_I(a_{1,i}, a_{2,i}))_{i=1}^{k1} \oplus (a_i)_{i=k1+1}^{k2}$.
  - For all other cases, inserting $b \in B \cup I$ in a stack layer $m \times (a_i)_{i=1}^k$: $\mathcal{I}_I(m \times (a_i)_{i=1}^k, b) = m \times (b \oplus (a_i)_{i=1}^k)$. The new constraint is perpended to the front of the list of existing constraints.

The stack is defined to be a sequence of layers:

$$q_0 \mid q_1 \mid ... \mid q_{N-1} \mid \ q_N,\ q_i \in I$$

q_0 is the bottom layer and q_N is the top layer. The `push` operation of the stack is performed by adding a new empty layer $1 \times [\ ]$ to the top of the stack whenever the analysis enters a new scope. The `pop` operation is performed on exiting a scope by merging the top two layers of the stack using the $\mathcal{I}_I$ operator as follows:

$$\texttt{pop}(q_0 \mid q_1 \mid ... \mid q_{N-1} \mid \ q_N) = q_0 \mid q_1 \mid ... \mid \mathcal{I}_I(q_{N-1}, q_N)$$

The stack concretization function γ_{STACK} is defined as follows: $\gamma_I(q_0) \mid \gamma_I(q_1) \mid ... \mid \gamma_I(q_N)$. The binary operators join, meet, widening operators are applied pairwise to the layers of the stack. The backward assignment operator is applied to every layer of the stack.

2.2.3 Example Sub-domains

[Examples of sub-domains plugged into our analysis.]

3 Implementation

4 Evaluation

Bibliography

[1] David M Blei and Padhraic Smyth. Science and data science. *Proceedings of the National Academy of Sciences*, 114(33):8689–8692, 2017.

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

[3] 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.

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

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

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

[7] 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.