Automated Checking of Implicit Assumptions on Textual Data

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

August 24, 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 Deoxyribonucleic acid (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 these reasons, 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 with respect to a given data science program. Unlike the data-cleaning tools mentioned above, our method does not perform computations on the dataset directly. Instead, we design a static analysis that analyzes the source code of the program in order to find out the implicit assumptions that the program makes about the input data that it reads such that if these assumptions are violated the program will crash. These assumptions represent the pre-conditions that must

be satisfied so that the program runs without producing errors. Our analysis will compute an over-approximation of these pre-conditions. In other words, if there is a set of accepted values that will allow the program to run without raising an error, the set of values that our analysis computes will contain all the accepted values, but it might contain some extra values that will cause the program to raise errors. That is, the pre-conditions we compute are necessary but not sufficient for the program to terminate without crashing.

We then design an input checker that looks for values that violate the assumptions computed by the analysis. If that is the case, the wrong input values are flagged as errors so that the user can correct them. Our approach is advantageous in cases where the data cannot be shared with third parties to perform data cleaning. We implement our analysis and input checker for Python programs and integrate them into a tool that can be used directly by domain experts on their data.

```
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

To elaborate further on what we aim to achieve in this thesis, 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 (line 1), 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 (lines 2-5). 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? For our analysis to compute a successful over-approximation, it must always include the character 'A', 'C', 'G' and 'T' in the set of accepted values it produces. However, it may include some other values as well.

In section 1.1, we present the theoretical aspects upon which the work of this thesis is based. In section 2, we explain the design of our static analysis and its different components in more detail. In section 3, we give an overview of how our analysis and input checker were implemented and integrated into an IDE plug-in. We evaluate the work achieved in this thesis in section 4 and, finally, the conclusion and possible future work are to be found in section 5.

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 and Contribution

We build our approach on the work done in [4]. This work was focused on inferring assumptions on numerical values in terms of types, ranges and simple relations. They designed a type domain which we also use in this thesis and describe in section 2.3.1 and a simple relations domain to keep track of relations between numerical variables such as $x \leq y$. For ranges the used the well-known Interval Domain [5]. They also used a stack mechanism to store assumptions collected about the input values by the domains they employed.

Since we would like to analyze more evolved code examples that contain assumptions on both numerical and string values, such as Listing 1.1, we take a more generic approach to the problem in this thesis. We design our static analysis to be a generic framework which can employ any existing value domain to keep track of assumptions on the input data. This way, we can employ more powerful domains, for example the Octagon Domain which we use in this thesis, to keep track of relations between numerical variables and we can also keep track of assumptions on different types of variables, for instance numerical and string variables, at the same time. We also extend the stack mechanism used in

[4] to be more generic and to keep track of assumptions from different domains simultaneously.

2 Static Analysis Design

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 string values.
- $\mathcal{V}$: the set of all program variables.
- $\wp(\mathcal{V} \to \mathcal{S})$: the set of environments mapping program variables to their possible values.
- $\wp(\mathcal{S})^*$: 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. Specifically, given a program, its concrete semantics is the set of values that its variables are allowed to take and the list of input values that it is allowed to read so that it terminates without 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 by starting at the final

state of the program and analyzing all the execution paths of the program till it reaches the initial state.

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. We define a $SUBD$ to be a family of domains that can be used in our analysis. Any value domain can be a member of this family given that it define extra operators that are described in 2.2.1. The Assumption Domain uses these domains to keep track of assumptions on the values of program variables. For example, in the case of , 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*.

Going back to Listing 2, if we trace a backward static analysis from line 24 to line 15 using the Character Inclusion Domain as a sub-domain, we will get that the variable *base* is allowed to contain only the characters 'A', 'C', 'G' and 'T'. When this variable is read as input on line 14, we need to store the assumption in some data structure. The same applies for the variable $sequence\_length$. Tracing the analysis backwards using the Octagon Domain will give us the constraint $sequence\_length \leq max\_length$. On line 6, when the variable is read as input, this constraint must be stored in a way that preserves its order among other inputs that the program reads in the course of its execution. More specifically, we need a data structure in which the assumptions about $sequence\_length$ appear before those about *base*. We also need a data structure that indicates how many times an input value is read. For example, it needs to show that the variable *base* is read $sequence\_length$ number of times.

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 branch or a loop, a new layer is pushed onto the stack. Inside of this scope, whenever a variable is read as input, we store its constraints in the top layer of the stack. When exiting a branch or a loop, the top layer is popped from the stack and combined with the layer below it in such a way that the assumptions appear in the order in which the inputs are read. In the case of exiting a loop, we define a mechanism to indicate how many times a group of assumptions are repeater. This way, at the end of the analysis, the stack will contain all the assumptions on the input value of the program in the correct order. We define the stack with further detail in section 2.2.2.

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

We define $STACK$ to be a set of all possible stacks which store assumptions on input values of the program.

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

We can 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.
- A concretization function $\gamma_D(d) = (\bigcap_{i=1}^{n} \gamma_{S_i}(S_i), \gamma_{STACK}(Q))$.
- 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 small than or equal to another element d_1 if and only if d_1 represents a set of allowed input values that is smaller than or equal the set allowed input values by d_2.
- A minimum element $\bot_D = ((\bot_{S_i})_{i=1}^n, \bot_{STACK})$. This elements represents a state where there are no possible input values that are acceptable by the program.
- A maximum element $\top_D = ((\top_{S_i})_{i=1}^n, \top_{STACK})$. This represents the set of all possible input value without any constraints.
- A join operator $\sqcup_D$ such that $((S_{1,i})_{i=1}^n, Q_1) \sqcup_D ((S_{2,i})_{i=1}^n, Q_2) = (((S_{1,i} \sqcup_{S_i} S_{2,i})_{i=1}^n, Q_1 \sqcup_{STACK} Q_2))$. When joining two paths of the analysis, we must join the assumptions computed by the analysis on the program variables.
- 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 that represent an input value. 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 *replacement* operator $\mathcal{R}_F(v, f, x)$ that, when an input value is read and stored in variable $v \in \mathcal{V}$, introduces some special variable x into element f of a relational domain F that denotes this input value and its relation to other variables. This newly introduced variable has to provide information on the order in which an input value was read with respect to other input values in the program. One possible definition for a replacement operator is to replace a variable that is read as input with a special variable l_i that represents the program point i at which it is read. In this case the operator is $\mathcal{R}_F(v, f, l_i)$.

If a program reads inputs from two different paths and these paths are to be joined at some point in the analysis, relational domain elements resulting from these two paths will contain different variables representing the inputs read on these paths. For this reason, we require 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. From the perspective of our analysis, we do not care about the particular variable that represents an input, but rather about the order in which these input are being read in their respective paths. The unification operator needs to ensure that the constraints on two input value read in the same order on two different paths can be joined successfully. An illustrative example on the replacement and unification operators can be found in section 2.3.2 where we refer to the Octagon Domain.

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 prepended 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:

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

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.

2.3 An Instance of the Analysis

After designing an Assumption Domain in the previous section, we create an instance of this domain with several sub-domains for the purpose of this thesis. Here we introduce these sub-domains: the Type Domain, the Octagons Domain and the Character Inclusion Domain. We then trace an instance of the analysis instantiated with these three domains on a code example.

Similar to those defined in [8], we define an expression language that will be useful in defining substitution and filter algorithms for our sub-domains later in this section:

$$\begin{array}{lr} \text{expr} ::= \text{v} & (\text{variable, } v \in \mathcal{V}) \\ \quad \mid \text{c} & (\text{literal, } c \in \mathcal{S} \cup \mathbb{R}) \\ \quad \mid \text{expr} \diamond \text{expr} & (\text{expression, } \diamond \in \{+, -, \times \div\}) \end{array}$$

$$\begin{array}{lr} \text{cond} ::= \text{expr} = \text{expr} & (\text{comparison}) \\ \quad \mid \text{cond} \wedge \text{cond} & (\text{logical and}) \\ \quad \mid \text{cond} \vee \text{cond} & (\text{logical or}) \end{array}$$

2.3.1 The Type Domain

The Type Domain is used to keep track of types of variables. We define a type lattice **T** as per the Hass diagram below:

In section 2.1 we define $\mathcal{S}$ to be the set of all possible strings. Here we define the sets $\mathbb{F}, \mathbb{I}, \mathbb{B} \subseteq \mathcal{S}$ to be the sets of strings that can be interpreted as floating-point numbers, integers and booleans respectively. Then we define a concretization function $\gamma_{\mathbf{T}} : \mathbf{T} \longrightarrow \mathcal{S}$ for the type lattice as follows:

$$\begin{gathered}\gamma_{\mathbf{T}}(String) = \mathcal{S} \\ \gamma_{\mathbf{T}}(Float) = \mathbb{F} \\ \gamma_{\mathbf{T}}(Integer) = \mathbb{I} \\ \gamma_{\mathbf{T}}(Boolean) = \mathbb{B} \\ \gamma_{\mathbf{T}}(\bot) = \phi\end{gathered}$$

The operators $\sqsubseteq_{\mathbf{T}}, \sqcup_{\mathbf{T}}, \sqcap_{\mathbf{T}}$ can be defined using the Hass diagram above.

We define arithmetic operations for the type lattice as follows:

$$t_1 + t_2 = \begin{cases} Integer & t_1 = Boolean \wedge t_2 = Boolean \\ String & t_1 = String \wedge t_2 = String \\ \bot_{\mathbf{T}} & (t_1 = String \wedge t_2 \neq String) \vee (t_1 \neq String \wedge t_2 = String) \\ t_1 \sqcup_{\mathbf{T}} t_2 & otherwise \end{cases}$$

$$t_1 - t_2 = \begin{cases} Integer & t_1 = Boolean \wedge t_2 = Boolean \\ \bot_{\mathbf{T}} & t_1 = String \vee t_2 = String \\ t_1 \sqcup_{\mathbf{T}} t_2 & otherwise \end{cases}$$

$$t_1 \times t_2 = \begin{cases} Integer & t_1 = Boolean \wedge t_2 = Boolean \\ \bot_{\mathbf{T}} & t_1 = String \wedge (t_2 = String \vee t_2 = Float) \\ \bot_{\mathbf{T}} & t_2 = String \wedge (t_1 = String \vee t_1 = Float) \\ t_1 \sqcup_{\mathbf{T}} t_2 & otherwise \end{cases}$$

$$t_1 \div t_2 = \begin{cases} \bot_{\mathbf{T}} & t_1 = Boolean \vee t_2 = Boolean \\ Float & otherwise \end{cases}$$

To define the Type Domain, we assume that a static type inference has already been run on the program and that it infers the most generic type for a variable in all execution paths of the program. Our Type Domain aims to track

more fine-grained information about the types of variables. We assume that the function $\texttt{type}(v)$ returns the type of a variable according to the type inference. Then we proceed to define the Type Domain as follows:

$$\text{TYP} \equiv \mathcal{V} \longrightarrow \mathbf{T}$$

- An element $t \in \text{TYP} = \{v \to typ \mid v \in \mathcal{V} \wedge typ \in \mathbf{T}\}$. An element of this domain maps a variable to an element of the type lattice.
- A concretization function $\gamma_{\mathbf{TYP}}$:
  - We first define a function $\gamma_c : (\mathcal{V} \to \wp(\mathcal{S})) \longrightarrow \wp(\mathcal{V} \to \mathcal{S})$, which transforms a mapping from variable to multiple string values to multiple mappings from a variable to a string value.
  - We define an intermediate concretization function $\gamma'_{\mathbf{TYP}} : \text{TYP} \longrightarrow (\mathcal{V} \to \wp(\mathcal{S}))$ as follows: $\gamma'_{\text{TYP}}(f) = \lambda_x \cdot \gamma_{\mathbf{T}}(f(x))$.
  - Finally, the concretization function of the Type Domain $\gamma_{\text{TYP}} : \text{TYP} \longrightarrow \wp(\mathcal{V} \to \mathcal{S})$ as the chaining of the two previous functions: $\gamma_c \circ \gamma'_{\text{TYP}}$. It maps every variable to the possible values it can take according to its type.
- A partial order $\sqsubseteq_{\text{TYP}}$: $t_1 \sqsubseteq_{\text{TYP}} t_2 \iff \forall_{v \in \mathcal{V}}(t_1(v) \sqsubseteq_{\mathbf{T}} t_2(v))$. For an element of the domain to be smaller than the other, it has to the case that it maps every variable involved in the program to a smaller lattice element than it is mapped in the other element.
- A minimum element $\bot_{\text{TYP}} = \lambda_x \cdot \bot_{\mathbf{T}}$. It simply maps every variable to the least element of the type lattice. This represents a type error state.
- A maximum element $\top_{\text{TYP}} = \lambda_x \cdot \texttt{type}(x)$. Since the type inference calculates the most generic type a variable can take, the maximum element of this domain maps every variable to the type computed for it by the type inference. If a variable has a type Float in the type inference, then this it the most generic it can have in all paths of the program and therefore we do not assign it to String but to Float.
- A join operator $\sqcup_{\text{TYP}}$: $t_1 \sqcup_{\text{TYP}} t_2 = \lambda_x \cdot t_1(x) \sqcup_{\mathbf{T}} t_2(x)$. If variable is mapped to two different types in two different paths, then, when these paths are joined, we map the variable to the more *generic* type. That is, the type that is higher in the type lattice Hass diagram.
- A meet operator $\sqcap_{\text{TYP}}$: $t_1 \sqcap_{\text{TYP}} t_2 = \lambda_x \cdot t_1(x) \sqcap_{\mathbf{T}} t_2(x)$.
- A backward assignment operator $[\![X := aexpr]\!](t) = \text{SUBS}_{\text{TYP}}(t, X, aexpr)$ which is described in Algorithm 1. The substitution algorithm evaluates *aexpr* and all its sub-expressions and then refines the state t using this evaluation and the type of X.
- A filter operator $[\![cond]\!](t) = t$. It does not affect the state.

- A widening operator $\nabla_{\text{TYP}} \equiv \sqcup_{\text{TYP}}$.

  We note that since the Type Domain is a non-relational domain, a replacement operator $\mathcal{R}_{\text{TYP}}$ and a unification operator $\mathcal{U}_{\text{TYP}}$ return the state unchanged.

Algorithm 1 Substitution algorithm for Type Domain

function $\text{SUBS}_{\text{TYP}}(t,\ x,\ aexpr)$
  $value \leftarrow t(x)$
  $t(x) \leftarrow \texttt{type}(x)$
  $eval \leftarrow empty\ map$
  $eval \leftarrow \text{EVAL}_{\text{TYP}}(aexpr, eval)$
  $\text{REFINE}_{\text{TYP}}(aexpr, eval, eval[x] \sqcap_{\mathbf{T}} value, t)$
  return t

Algorithm 2 Expression evaluation for Type Domain

function $\text{EVAL}_{\text{TYP}}(expr, eval)$
  if $expr$ is a literal **then**
    $eval[expr] \leftarrow$ type of literal
    return $eval$
  if $expr \in \mathcal{V}$ **then**
    $eval[expr] \leftarrow \texttt{type}(v)$
    return $eval$
  if $expr = expr1 \diamond expr2$ **where** $\diamond \in \{+, -, \times, \div\}$ **then**
    $eval[expr] \leftarrow \text{EVAL}_{\text{TYP}}(expr1) \diamond \text{EVAL}_{\text{TYP}}(expr2)$
    return $eval$

Algorithm 3 Expression refinement for Type Domain

function $\text{REFINE}_{\text{TYP}}(expr, eval, value, t)$
  if $expr$ is a literal **then**
    return t
  if $expr \in \mathcal{V}$ **then**
    $t(x) \leftarrow eval[expr] \sqcap_{\mathbf{T}} value$
    return t
  if $expr = expr1 \diamond expr2$ **where** $\diamond \in \{+, -, \times, \div\}$ **then**
    $value \leftarrow eval[expr] \sqcap_{\mathbf{T}} value$
    $refine1 \leftarrow \text{REFINE}_{\text{TYP}}(expr1, eval, value, t)$
    $refine2 \leftarrow \text{REFINE}_{\text{TYP}}(expr2, eval, value, refine1)$
    return $refine2$

2.3.2 The Octagon Domain

The Octagon Domain was introduced by Antoine Miné in [6] as a numerical relational domain that can keep track of relations the form $\pm X \pm Y \geq c$ where X and Y are program variables. We use the Octagon Domain in our analysis in order to keep track of relations between variables such as $sequence\_length \leq max\_length$ in Listing 2.1.

As explained in section 2.2.1, in order for an existing domain to work with our analysis framework, it needs to define a *replacement* operator that, whenever an input is read, introduces a new variable to denote this input and the constraints and relations associated with it before it is stored in the stack. In the same section, we also explain the need to define a unification operator to address the problem of inconsistent environments arising from introducing new variables into our analysis.

```
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
```

Listing 2.2: Example of Unification

We define a replacement operator $\mathcal{R}_O(o, v, l_i)$ for the Octagon Domain, where o is an octagon, v is the variable that is being read as input and l_i is the program point at which the input is being read. The operator works by first adding the variable l_i to the octagon. This variable has no constraints associated with it at first. Then, assuming the Octagons Domain defines a backward assignment operator, we perform the substitution $[\![v := l_i]\!]_O$. Thus, v is replaced by l_i in all the constraints of the octagon and v becomes top.

We define the unification operator $\mathcal{U}_O(o_1, o_2)$ where both o_1 and o_2 are octagons. The operator is applied before every join, meet or widening operation in the analysis. We assume every element of the Octagon Domain keeps a list of its variables in the order in which they were added during the analysis. For every variable l_i in o_1 and l_j in o_2, that have the same order in the respective variable lists, if $j < i$, we replace l_i with l_j in o_1. That is, when we have two variables that represent inputs read in the same order in two different paths in the analysis, we replace the variable that represents a later program point, with a variable that represents an earlier program point. After this step, if there are variables in o_2 which are not in o_1, they are added to o_1. Note that in this case the operator is asymmetric and needs to be called twice, $\mathcal{U}_O(o_1, o_2)$ and $\mathcal{U}_O(o_2, o_1)$, before every join, meet and widening.

To illustrate the replacement and unification operators we introduce Listing 2.2. 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 $z + x > 20$. If we trace a backward analysis on this program and employing the $\mathcal{R}_O$ we defined earlier in this section, then in the then-branch, we get $\mathcal{R}_O(\{y + x > 10\}, y, l_3) = \{l3 + x > 10\}$ and similarly in the else-branch we get the relation $\{l_7 + x > 20\}$. A join needs to be performed on these two elements at the head of the if-statement. Since our analysis treats l_3 and l_7 as different variables, without the unification operator, the join will yield top.

As mentioned earlier, our analysis does not care about the specific variables representing input values, but rather about the order in which they occur in the list of input values which the program reads in their respective paths. In this example, we only care that after reading x we read another variable whose summation to x had to be greater than 20, otherwise the program will crash. Therefore, for the purpose of our analysis, l_3 and l_7 should be treated as the same variable when joining the two paths at the head of the if-condition since both of them are read second in their respective paths. We call the unification operator once as follows: $\mathcal{U}_O(\{l_3 + x > 10\}, \{l_7 + x > 20\}) = \{l_3 + x > 10\}$ and then another time switching the operands as follows: $\mathcal{U}_O(\{l_7 + x > 20\}, \{l_3 + x > 10\}) = \{l_3 + x > 20\}$. Then a join can be performed between $\{l_3 + x > 10\}$ and $\{l_3 + x > 20\}$ which will give us $l_3 + x > 20$.

2.3.3 The Character Inclusion Domain

The Character Inclusion Domain, as defined in [7], is an abstract domain that maps a string into two sets: a set of characters that are certainly included in the string and another set of characters that may be included in the string. For a finite alphabet A, we define a character inclusion lattice as follows:

- An element $c \in CI = \{(C, M) : C, M \in \wp(\mathrm{A}) \wedge C \subseteq M\} \cup \bot_{CI}$. This can be understood as an abstraction of a string which certainly contains all the characters in C and is only allowed to contain characters in M.

- A concretization function γ_{CI}: $\langle CI, \sqsubseteq_{CI}\rangle \longrightarrow \langle A^*, \subseteq\rangle$ such that $\gamma_{CI}((C, M)) = \{x \mid x \in A^* \wedge \texttt{char}(x) \supseteq C \wedge \texttt{char}(x) \subseteq M\}$, where $\texttt{char}(x)$ is a function that returns the set of all characters in a string x. An element (C, M) concertizes to the set of all string which contain all the characters in C and which do not contain any characters outside M.

- A partial order: $(C_1, M_1) \sqsubseteq_{CI} (C_2, M_2) \iff (C_1 \supseteq C_2, M_1 \subseteq M_2)$. This order indicates that the fewer the certainly included characters and the more the maybe included characters, the more strings we represent. For example the element $(\{a\}, \{a, b\})$ concertizes to $\{a, aa, ab, aaa, aab, ...\}$ while the element $(\emptyset, \{a, b\})$ concertizes to $\{\varepsilon, a, b, aa, ab, bb, ...\}$. The restriction of having a in the certainly included set makes the values $\{\varepsilon, b, bb, bbb, ...\}$ disappear from the concretization.

- A least element $\bot_{CI}$ which represents a failure state.

- A greatest element $\top_{CI} = (\emptyset, A)$ which represents a string that can contain any combination of characters from the alphabet.

- A join operator: $(C_1, M_1) \sqcup_{CI} (C_2, M_2) = (C_1 \cap C_2, M_1 \cup M_2)$. The soundness of this operator can be stated as follows:
  $\{x \mid x \in A^* \wedge \texttt{char}(x) \supseteq C_1 \wedge \texttt{char}(x) \subseteq M_1\} \cup \{x \mid x \in A^* \wedge \texttt{char}(x) \supseteq C_2 \wedge \texttt{char}(x) \subseteq M_2\} \subseteq \{x \mid x \in A^* \wedge \texttt{char}(x) \supseteq C_1 \cap C_2 \wedge \texttt{char}(x) \subseteq M_1 \cup M_2\} \iff \gamma_{CI}((C_1, M_1)) \cup \gamma_{CI}((C_2, M_2)) \subseteq \gamma_{CI}((C_1, M_1) \sqcup_{CI} (C_2, M_2))$.
  If we have two elements $(\{a, b\}, \{a, b, c\})$ and $(\{a\}, \{a, b, d\})$ that we would like to join, then we can only say that the resulting element will *certainly* contain the character a (the intersection of C sets), but it *may* contain any of the characters $a,\ b,\ c$ or d (the union of the two M sets).

- A meet operator: $(C_1, M_1) \sqcap_{CI} (C_2, M_2) = (C_1 \cup C_2, M_1 \cap M_2) \cup \bot_{CI}$. The bottom is to account for cases when $C_1 \cup C_2 \not\subseteq M_1 \cap M_2$. The soundness of this operator can be stated as follows:
  $\{x \mid x \in A^* \wedge \texttt{char}(x) \supseteq C_1 \wedge \texttt{char}(x) \subseteq M_1\} \cap \{x \mid x \in A^* \wedge \texttt{char}(x) \supseteq C_2 \wedge \texttt{char}(x) \subseteq M_2\} \subseteq \{x \mid x \in A^* \wedge \texttt{char}(x) \supseteq C_1 \cup C_2 \wedge \texttt{char}(x) \subseteq M_1 \cap M_2\} \iff \gamma_{CI}((C_1, M_1)) \cap \gamma_{CI}((C_2, M_2)) \subseteq \gamma_{CI}((C_1, M_1) \sqcap_{CI} (C_2, M_2))$.
  The meet works in the opposite way to the join. If we have two elements $(\{a, b\}, \{a, b, c\})$ and $(\{a\}, \{a, b, d\})$ which we would like to meet, we can say that the resulting element will *certainly* contain both a and b (the union of the two C sets) and that it *may* contain only a and b (the intersection of the two M set) as well.

- A concatenation operator: $((C_1, M_1) +_{CI} (C_2, M_2)) = ((C_1 \cup M_1), (C_2 \cup M_2))$. When concatenating two string, the resulting string will *certainly* that characters that are certainly included in both strings and *may* contain the characters that may be included in both strings as well.

- A widening operator $\nabla_{CI} \equiv \sqcup_{CI}$. The widening operator is the same as the join.

Then we define the Character Inclusion Domain as a mapping from variables to elements of the character inclusion lattice as follows:

$$\text{CHAR} \equiv \mathcal{V} \longrightarrow CI$$

- An element $c \in CHAR = \{v \longrightarrow ci \mid v \in \mathcal{V} \wedge ci \in CI\}$. Every element in the Character Inclusion Domain maps a variable to an element in the character inclusion lattice.

- A concretization function γ_{CHAR}:
  - We first define a function $\gamma_c : (\mathcal{V} \to \wp(\mathcal{S})) \longrightarrow \wp(\mathcal{V} \to \mathcal{S})$, which transforms a mapping from variable to multiple string values to multiple mappings from a variable to a string value.
  - We define an intermediate concretization function $\gamma'_{\text{CHAR}} : \text{CHAR} \longrightarrow (\mathcal{V} \to \wp(\mathcal{S}))$ as follows: $\gamma'_{\text{CHAR}}(f) = \lambda_x \cdot \gamma_{CI}(f(x))$.

– Finally, we define the concretization function of the Character Inclusion Domain $\gamma_{\text{CHAR}} : \text{CHAR} \longrightarrow \wp(\mathcal{V} \to \mathcal{S})$ as the chaining of the two previous functions: $\gamma_c \circ \gamma'_{\text{CHAR}}$.

- A partial order $\sqsubseteq_{\text{CHAR}}$: $c_1 \sqsubseteq_{\text{CHAR}} c_2 \iff \forall_{v \in \mathcal{V}}(c_1(v) \sqsubseteq_{CI} c_2(v))$.
- A minimum element $\bot_{\text{CHAR}} = \lambda_x \cdot \bot_{CI}$. Every variable is mapped to the smallest element in the character inclusion lattice.
- A maximum element $\top_{\text{CHAR}} = \lambda_x \cdot \top_{CI}$. Every variable is mapped to the biggest element of the character inclusion lattice.
- A join operator $\sqcup_{\text{CHAR}}$: $c_1 \sqcup_{\text{CHAR}} c_2 = \lambda_x \cdot c_1(x) \sqcup_{CI} c_2(x)$ which maps a variable to the joining of the two lattice elements to which it was mapped in c_1 and c_2.
- A meet operator $\sqcap_{\text{TYP}}$: $c_1 \sqcap_{\text{CHAR}} c_2 = \lambda_x \cdot c_1(x) \sqcap_{CI} c_2(x)$ which maps a variable to the meet of the two lattice elements to which it was mapped in c_1 and c_2.
- A backward assignment operator $[\![X := aexpr]\!](c) = \text{SUBS}_{\text{CHAR}}(c, X, aexpr)$ which is described in Algorithm 4. It evaluates *aexpr* and all its subexpressions and refine the state c with the values computed in this evaluation and the value to which X was mapped before the assignment is invoked. Meanwhile, X becomes top.
- A filter operator $[\![cond]\!](c) = \text{FILTER}_{\text{CHAR}}(cond, c)$ which is described in Algorithm 5.
- A widening operator $\nabla_{\text{CHAR}} \equiv \sqcup_{\text{CHAR}}$ which is equivalent to the join.

As is the case for the Type Domain, the replacement operator $\mathcal{R}_{\text{CHAR}}$ and a unification operator $\mathcal{U}_{\text{CHAR}}$ do not produce any changes on the state since this domain is non-relational.

Algorithm 4 Substitution algorithm for Character Inclusion Domain

```
function SUBS_CHAR(c, x, aexpr)
    value ← c(x)
    c(x) ← ⊤_CI
    eval ← emtpy map
    eval ← EVAL_CHAR(aexpr, eval, c)
    REFINE_CHAR(aexpr, eval, eval[x] ⊓_CI value, c)
    return c
```

Algorithm 5 Filter algorithm for Character Inclusion Domain

function $\text{FILTER}_{\text{CHAR}}(cond,\ c)$
  if $cond = (cond1 \wedge cond2)$ **then**
    $c1 \leftarrow \text{FILTER}_{\text{CHAR}}(cond1, c)$
    $c2 \leftarrow \text{FILTER}_{\text{CHAR}}(cond2, c)$
    return $c1 \sqcap_{\text{CHAR}} c2$
  else if $cond = (cond1 \vee cond2)$ **then**
    $c1 \leftarrow \text{FILTER}_{\text{CHAR}}(cond1, c)$
    $c2 \leftarrow \text{FILTER}_{\text{CHAR}}(cond2, c)$
    return $c1 \sqcup_{\text{CHAR}} c2$
  else if $cond = (expr1 == expr2)$ **then**
    $eval1 \leftarrow$ *empty map*
    $eval1 \leftarrow \text{EVAL}_{\text{CHAR}}(expr1, eval1, c)$
    $eval2 \leftarrow$ *empty map*
    $eval2 \leftarrow \text{EVAL}_{\text{CHAR}}(expr2, eval2, c)$
    $\text{REFINE}_{\text{CHAR}}(expr1, eval1, eval1[expr1], c)$
    $\text{REFINE}_{\text{CHAR}}(expr2, eval2, eval2[expr2], c)$
  return c

Algorithm 6 Expression evaluation for Character Inclusion Domain

function $\text{EVAL}_{\text{CHAR}}(expr, eval, c)$
  if $expr$ is a literal **then**
    $eval[expr] \leftarrow (\texttt{char}(x), \texttt{char}(x))$
    return $eval$
  if $expr \in \mathcal{V}$ **then**
    $eval[expr] \leftarrow c(expr)$
    return $eval$
  if $expr = expr1 + expr2$ **then**
    $eval[expr] \leftarrow \text{EVAL}_{\text{CHAR}}(expr1) +_{CI} \text{EVAL}_{\text{CHAR}}(expr2)$
    return $eval$

Algorithm 7 Expression refinement for Character Inclusion Domain

function $\text{REFINE}_{\text{CHAR}}(expr, eval, (C, M), c)$
  if $expr$ is a literal **then**
    return c
  if $expr \in \mathcal{V}$ **then**
    $c(x) \leftarrow eval[expr] \sqcap_{CI} (C, M)$
    return c
  if $expr = expr1 + expr2$ **then**
    $(C, M) \leftarrow eval[expr] \sqcap_{CI} (C, M)$
    $refine1 \leftarrow \text{REFINE}_{\text{TYP}}(expr1, eval, (\emptyset, M), c)$
    $refine2 \leftarrow \text{REFINE}_{\text{TYPE}}(expr2, eval, (\emptyset, M), refine1)$
    return $refine2$

3 Implementation

We implemented our static analysis and input checker in Python as an extension to the Lyra project [9]. We then implemented a plug-in for IntelliJ IDEA that enables the user to use the analysis and checker as a tool. The plug-in implementation is in Java.

We implement the Assumption Domain described in section 2.2 in a class we call *AssumptionState*. The *AssumptionState* class has a list of sub-states which represents the sub-domains in the Assumption Domain. The Type Domain already exists as part of Lyra. We implemented separate classes for the Octagon and Character Inclusion domains. For the Octagon domain, we used the Python interface of the ETH Library for Numerical Analysis (ELINA) [10]. We added extra logic in order to translate the Lyra representation of expressions and conditions into a representation that is compatible with ELINA and also to implement the replacement and unification operators mentioned in section 2.3.2. The Character Inclusion domain is implemented as described in section 2.3.3 with the addition of support for built-in python functions that perform checks on the characters included in a string such as `isalpha()`, `isalnum()`, `isdecimal()`, `isdigit()`, `islower()` and `isupper()`. We also added the Sign Domain, which had already exited as a part of Lyra, to our implementation in order to keep track of constraints such as $x \neq 0$ which is impossible to track with Octagons.

Every one of the sub-states keeps track of all program variables. If a variable belongs to a type that the domain cannot handle, then its value will always be top in this domain. Therefore, an integer variable will always be mapped to top in the Character Inclusion Domain and a string variable will never have any constraints on it in the Octagon Domain. We require all sub-states to define functions that convert their constraints into JSON object and parse them back from JSON object. We also require every sub-state to implement an input-checking function that, given a input value read from a file, is able to tell whether this value satisfies the constraints that the sub-state represents or not.

The class *InputStack* also has the stack described in section 2.2.2 which implemented as an extension to the already existing *Stack* interface in Lyra. We define the pop and push operators for it as well as the structure of individual stack layers. The stack layer is implemented in the class *InputLattice* which represents a list of possibly repeated constraints on the input data (see the set I in section 2.2.2) and defines a mechanism for recording assumptions on the

top layer as well as for joining lists of assumptions from different paths of the analysis.

Our implementation follows a hierarchy similar to that used in [4]. We implement an *AssumptionController* class which acts as the entry point for our tool and takes as input the path to a Python code file and an input data file. This class controls three other classes that are in the layer right below it in the hierarchy: *AssumptionAnalysis* which extends the Lyra *Runner* class, *JSONHandler* and *InputChecker*. The *AnalysisAnalysis* class is responsible for running a backward static analysis on a given Python program. The *JSONHandler* class is responsible for writing the assumptions resulting from the analysis into a JSON file and later reading them back from the JSON file. The *InputChecker* runs a checking algorithm an a given input file and produces a list of errors present in the file.

The analysis starts with an instance of the *AssumptionState* class set to top and traverses the program paths backward calling the appropriate functions from the class at every step. The final result of the analysis is stored in the stack of the final state and it is of type *InputLattice*. When the analysis terminates, the controller retrieves the result and passes it to the *JSONHandler* class which writes the assumptions into JSON file by calling the functions defined in every sub-state. For subsequent runs of the controller, it checks if the code of the program has been modified and if it has not been modified, it does not run the analysis again but calls the *JSONHandler* to read the assumptions for that program from the existing JSON file and convert them back into an *InputLattice* object. The assumptions are then passed to the *InputChecker* class along with the input file which is required to be checked for errors. The *InputChecker* traverses the input file line by line and checks the value in every line against the corresponding assumption by calling the input-checking functions defined in every sub-state. It produces a dictionary which maps a line number in the input file to a *CheckerError* object which contains an appropriate error message describing how the value on this line violates an assumption.

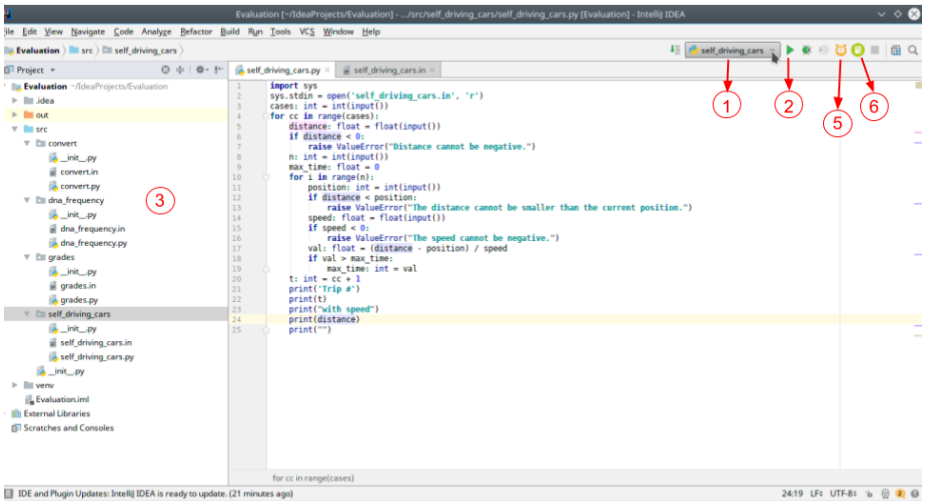


Figure 3.1: Screenshot of the IntelliJ IDEA Interface with the Plug-in

We implement a plug-in for IntelliJ IDEA that allows a user to run our analysis and input checker. The decision to implement such a plug-in was based on feedback received from participants in the user study conducted in [4]. The interface which is presented to the user is shown in Figure 3.1. To run the tool on a specific Python program, we must create a run configuration for it and select it from the drop-down menu labeled 1 in the figure. Using the green button labeled 6 we can choose which input data file to check. Then we can run the tool via the button labeled 5 in the figure. The lines which contain errors in the input file will be underlined in red and when the user hovers them with the mouse, an error message will appear. In addition to that, a pop-up message will always appear at the bottom of the screen indicating the location and error message of the first error value in the file. This functionality was added to guide the user step by step in fixing the errors in the file from top to bottom. If the tool cannot detect any more errors in the file, a green pop-up appears at the bottom of the screen indicating so.

4 Evaluation

In this section we present the evaluation of the work done in this thesis. In section 4.1, we present methodology and result of a user study we conducted in order to evaluate the usability of our tool. In section 4.2, we present an evaluation of the static analysis we defined in section 2.

4.1 User Study

We conducted a user study to evaluate the usability of the tool we developed. We asked ten users, five who have a background in computer science and five who study or work in other fields, to use our tool and give us feedback on how helpful they find it. The methodology of the user study is presented in the next section and the results are presented in section 4.1.2.

4.1.1 Methodology

In our user study, we asked our users to do two experiments. For every experiment we presented the user with a programs, accompanied by an input data file which contains some errors that will cause the program to crash, and asked them to try to fix the errors in the data file so that the program runs without crashing. For one experiment they were asked to do this without any help and for the other they were asked to do this with the help of our tool. In the sections below we describe the programs and the input data we use and the procedure of the two experiments.

Programs and Input Data

We used four programs in our user study. The programs 'Grades' (Listing 4.1) and 'Self-Driving Cars' (Listing 4.3) are taken from the user study conducted in [4]. The programs 'Convert' (Listing) and 'DNA Frequency' we obtained from the online sources [cite] and [cite] respectively. The code of the programs so that it can be handled by our analysis. For example, we removed statements that contain lists and function call. We also added new statement which provide more interesting assumptions on these input data. We constructed the input datasets to contain an average of eight errors per file. The errors we added

can be classified into two main types: errors which appear as a result of typing mistakes, such as the number **6.5**, which represents a grade, being typed as **6..5** on line 32 in Listing 4.2, and errors which result from some garbage values being printed in the data file due to, for example, the conversion of data from one format to another such as the value **G**, which represents a DNA base, being printed as **..G** on line 6 in Listing 4.8.

Experiments

As mentioned earlier, we asked every participant in our user study to do two experiments, one after the other. In both experiments the user was presented with a Python program and a text file containing some input data which has errors that will cause the program to crash. They were asked to not change the code of the program but to change the input data, only adding or modifying lines without deleting them, such that the program runs without crashing. The users were presented with an online form that walked them through the experiments step by step. Before starting the experiments, the users were asked to answer some general questions in the form indicating their age, field of study, current occupation and their level of familiarity with programming on five-point Likert scale ranging from 1 to 5.

In both experiments, the form explained to them the task that they are required to perform and then asked them to choose one of the four programs based on their month of birth to ensure some randomness in the experiment. They were handed a sheet of paper containing a brief description of the task the program tries to achieve. In the experiment without the tool, they were given instructions on how to run a Python program in IntelliJ IDEA, while in the experiments with the tool, they were given instructions on how to run the tool and what kind of error messages they should expect to see. Then we allowed the user to work on the problem for eight minutes each. Both the form and the problem descriptions are shown in the appendix.

After every task, we asked the users to answer some questions in the form. For both experiments we asked if they had been able to fix the errors in the input data and how many minutes they spent on this task. For the experiment with the tool, we added a question about whether or not they were able to run the program without raising any errors. This is to account for the cases were the user was able to fix all the errors detected by the tool, but the tool failed to flag some erroneous data due to imprecision in the analysis.

These questions were followed by a group of questions on a five-point Likert scale in which the options were: Strongly Disagree, Disagree, Neutral, Agree, Strongly Agree. In both experiments, we asked how frustrated the users felt trying to solve the problems and how understandable they found the error messages printed by the program in the first experiment and displayed by the tool in the second experiment. We added two more questions in the second experiment asking them to rate how user-friendly they found the tool and whether they would rather fix the errors in a data file with or without the help of the

tool.

At the end of the form there were two open-ended questions that aimed to get feedback from the users in their own words. The first questions asked the users what they liked about the tool and the second questions asked them to suggest improvements that they think should be added to the tool. The form as well as the descriptions of the problems can be found in the appendix.

4.1.2 Results

General questions

The ages of the participants ranges from 19 to 31 with an average of 23.4. Five participants have backgrounds in computer science (one user used the term "informatics" in the form), while two participants have backgrounds in physics, two in environmental engineering and one in mathematics. Six of our participants are bachelor's students, two are master's students, one participant is a postdoctoral researcher and one is a teaching assistant.

```
1   subject: str = input()
2   homeworks: int = int(input())
3   students: int = int(input())
4   for i in range(students):
5     id: str = input()
6     sum: float = 0
7     max: float = 0
8     for j in range(homeworks):
9       grade: float = float(input())
10      best: float = float(input())
11      if grade < 0 or grade > best:
12      raise ValueError
13      sum: float = sum + grade
14      max: float = max + best
15
16    average: float = sum / max * 100.0;
17    print("Student ID:")
18    print(id)
19    print("Average homework grade:")
20    print(average)
21    grade: float = float(input())
22    best: float = float(input())
23    if grade < 0 or grade > best:
24      raise ValueError
25    final_grade: float = grade / best * 100.0
26    print("Final test grade:")
27    print(final_grade)
```

Listing 4.1: Code for Program 'Grades'

```
CS 101: Introduction to Computer Science
3
4
34-8342
9
10
12
12
11
10
45
50
34-14002
6
10
13
12
-9
10
51
50
31-1234
5.5
10
9
12
10#
10
-43
50
34-2373
6..5
10
11
12
7
10
48
```

Listing 4.2: Input Data for 'Grades'

```
cases: int = int(input())
for case in range(cases):
    distance: float = float(input())
```

```
  if distance < 0:
    raise ValueError("Distance cannot be negative.")
  n: int = int(input())
  max: float = 0
  for i in range(n):
    position: int = int(input())
    if distance < position:
      raise ValueError("Distance cannot be smaller than
     the current position.")
    speed: float = float(input())
    if speed < 0:
      raise ValueError("Speed cannot be negative.")
    val: float = (distance - position) / speed
    if val > max:
      max: int = val
  t: int = case + 1
  print("Trip:")
  print(t)
  print("with speed")
  print(distance / max)
```

Listing 4.3: Code for Program 'Self-Driving Cars'

```
4
10.5
3
4
40
6
45.5
12
50
120
5>>
25km
90
45
99km/h
-57
-123
76
130
1200
140
50
2
```

```
  60
  120
  30
  112
  200
  2
  100
  120
  170
```

Listing 4.4: Input Data for 'Self-Driving Cars

```
items: int = int(input())
if items == 0:
  raise ValueError
for i in range(items):
  name: str = input()
  weight: float = int(input())
  if weight <= 0:
    raise ValueError
  unit: str = input()
  if unit == 'pounds' or unit == 'lb' or unit == 'lbs':
    weight: float = weight * 453.592 * 1e-3
  elif unit == 'ounces' or unit == 'oz' or unit == 'oz.':
    weight: float = weight * 28.35 * 1e-3
  elif unit == 'grams' or unit == 'gms' or unit == 'g':
    weight: float = weight * 1e-3
  elif unit == 'kilograms' or unit == 'kilo' or unit == '
   kg':
    pass
  else:
    raise ValueError
  print("Item: " + name)
  print("weight:")
  print(weight)
  print("kg")
```

Listing 4.5: Code Program 'Convert'

```
  10#
  earphones
  300
  gramx
  stereo set
  0
  kg
```

```
  camera tripod
  1@
  qounds
  trolly suitcase
  30
  lbs
  laptop lenovo 5070
  1
  kgs
  laptop lenovo thinkpad
  2>>
  pounds
  12x envelope
  35
  gm.
  bic blue pens 4-piece set
  125
  grams
  travel compass
  50
  grams
  vacuum cleaner
  2#0
  kgs
```

Listing 4.6: Input Data for 'Convert'

```
sequences: int = int(input())
if sequences < 1:
  raise ValueError("Expecting at least one DNA sequence
  ")
for s in range(sequences):
  length: int = int(input())
  a: int = 0
  c: int = 0
  g: int = 0
  t: int = 0
  for i in range(length):
    base: str = input()
    if base == 'A':
    a: int = a + 1
    elif base == 'C':
    c: int = c + 1
    elif base == 'G':
    g: int = g + 1
    elif base == 'T':
```

```
      t: int = t + 1
      else:
      raise ValueError
    separator: str = input()
    if separator == '.' or separator == '#':
      pass
    else:
      raise ValueError
    print("Frequency of A nucleotide:")
    print(a)
    print("Frequency of C nucleotide:")
    print(c)
    print("Frequency of G nucleotide:")
    print(g)
    print("Frequency of T nucleotide:")
    print(t)
    CG_content: float = (c + g) / (a + t + c + g) * 100.0
    print("CG content: ")
    print(CG_content)
    print("%")
```

Listing 4.7: Code for Program 'DNA Frequency'

```
 3..
 10
A
O
C
 ..G
G
X
T
A
C
G
#
 15
C
G@
T
A
C
&C*
G
T
O
```

```
24  G
25  A
26  C
27  T) )
28  G
29  T
30  *
31  5
32  A
33  C
34  T
```

Listing 4.8: Input Data for ‘DNA Frequency’

4.2 Evaluation of the Static Analysis

5 Conclusion and Future Work

A Appendix

Lyra Checker Evaluation

Thank you for agreeing to take part in the user study of my bachelor's project.

* Required

1. **What is your age?** *

2. **What is your field of study?** *

3. **What is your current occupation?**
   Mark only one oval.
   - ◯ Bachelor's student
   - ◯ Master's student
   - ◯ PhD student
   - ◯ Other: _____

4. **On a scale from 1 to 5, how familiar are you with computer programming?** *
   Mark only one oval.

	1	2	3	4	5	
I have never programmed before	◯	◯	◯	◯	◯	I have a strong programmer background

Experiment I: Fixing input data

In this experiment you are presented with a Python program that reads some input data from a file and does some computations on them. Some of the input data is wrong and will cause the program to raise some errors.

We would like you to try to fix the input data so that the program runs without raising errors. You can assume that the program is correct. Please do NOT modify it.

Choose a program to run

To ensure some randomization to the experiment, please choose a program to run based on your month of birth:
January-March: convert
April-June: dna_frequency
July-September: grades
October-December: self_driving_cars

5. **Please select which problem you chose.** *

Mark only one oval.

- convert
- dna_frequency
- grades
- self_driving_cars

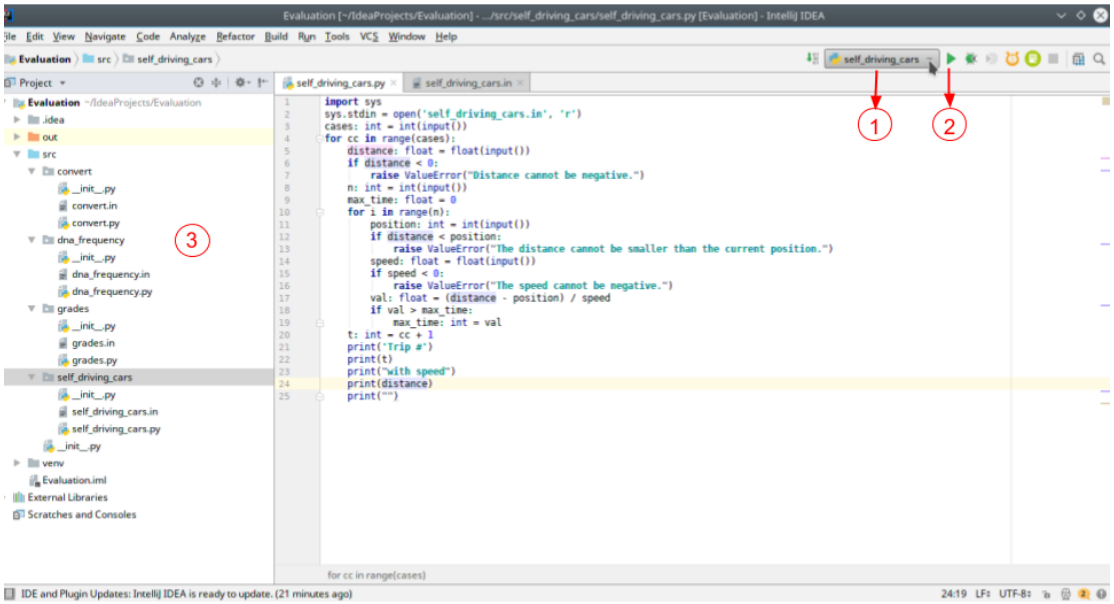
For the experiment to be successful, please read the instructions till the end of this page and the problem descriptions on paper carefully:

1- You will be using IntelliJ IDEA for this experiment. Each program is in a folder along with its input file. For example, the program convert.py reads input from the file convert.in. They both exist in the folder named "convert". You can view the input file data or program code by double-clicking them in the section Labeled 3 in the picture below.

2- To run a program, select its name form the drop down menu (Labeled 1) and click the Run button (Labeled 2).

3- You are given 8 minutes to try to correct as many errors as possible in the input data file of your chosen program.

IMPORTANT NOTE: You are allowed to look at the program code, but not to modify it. You can only add, or modify the lines of the input file, but not delete any of them completely.



Please tell me when you are ready to start.

Please answer these questions after finishing the task:

6. **How many minutes did this experiment take?** *

7. **Were you able to fix all the errors in the input data?** *
Mark only one oval.

- Yes
- No

Please rate how much you agree with the following statements:

8. **I felt frustrated trying to solve this problem** *
Mark only one oval per row.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
Rate					

9. **The errors messages printed by the program helped me identify what was wrong with the input** *
Mark only one oval per row.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
Rate					

Experiment II: Fixing input data with Lyra Checker

This experiment is similar to the previous one. But now you can use the Lyra Checker tool to help you locate wrong values in the input file.

Choose a program to run

Again, please choose a program to run based on your month of birth:
January-March: self_driving_cars
April-June: grades
July-September: dna_frequency
October-December: convert

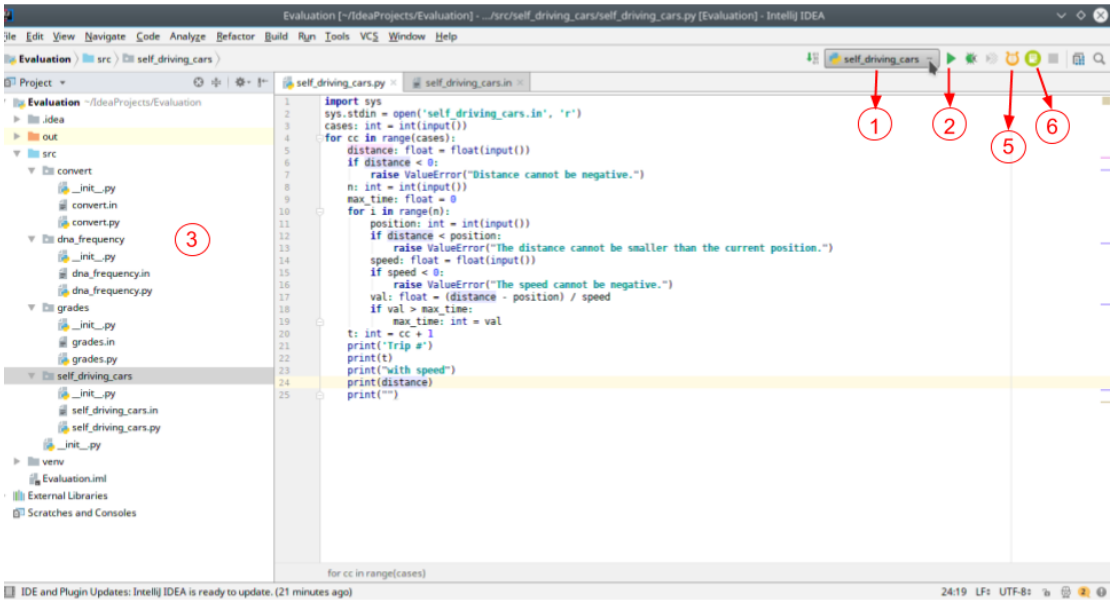
10. **Please select which problem you chose.** *
Mark only one oval.

- convert
- dna_frequency
- grades
- self_driving_cars

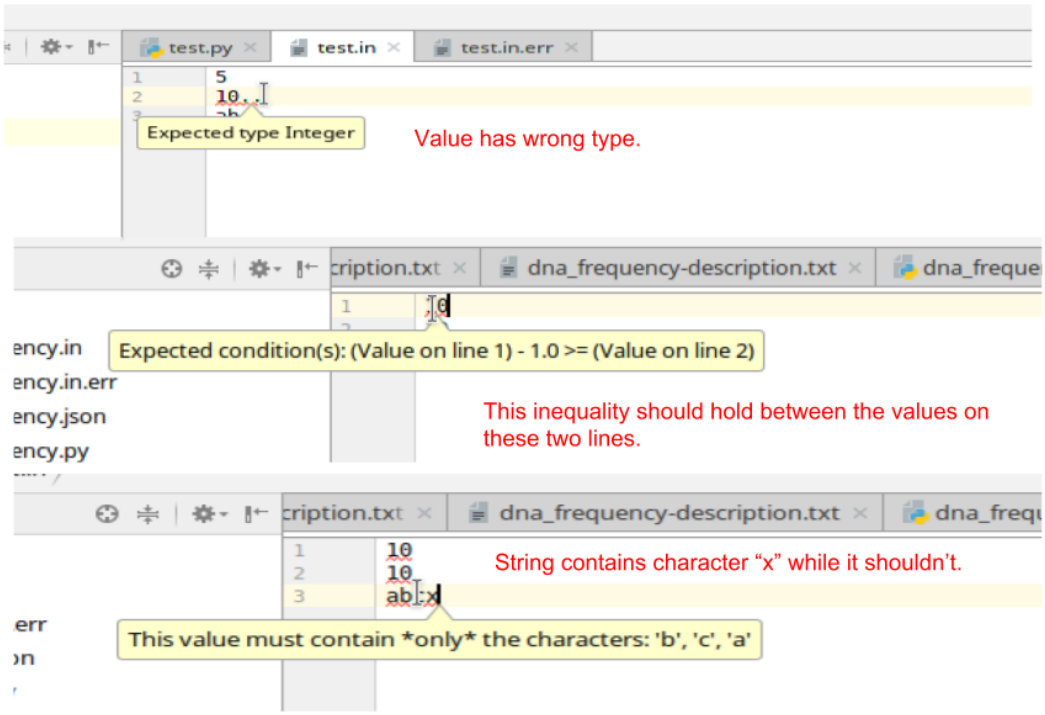
For the experiment to be successful, please read the instructions till the end of this page and the problem descriptions on paper carefully:

1- Choose the program from the drop-down menu labeled 1 in the picture below.
2- Choose the corresponding input file using the green button labeled 6.
3- Click the Lyra icon labeled 5 to run the tool and wait a few seconds.
4- Examine the error messages and try to fix the errors in the file. Note that the error messages will not be updated automatically once you change the data. You have to run the tool again to update them.
5- Repeat steps 3 and 4 until there are no more errors in the file.

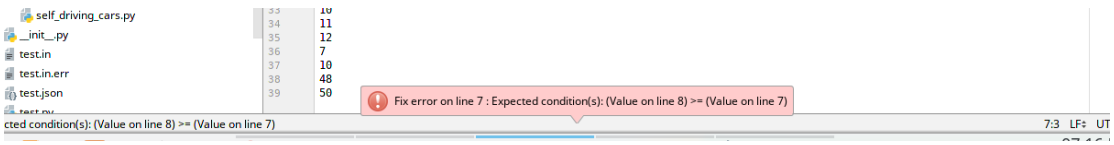
IMPORTANT NOTE: You are allowed to look at the program code, but not to modify it. You can only add, or modify the lines of the input file, but not delete any of them completely.



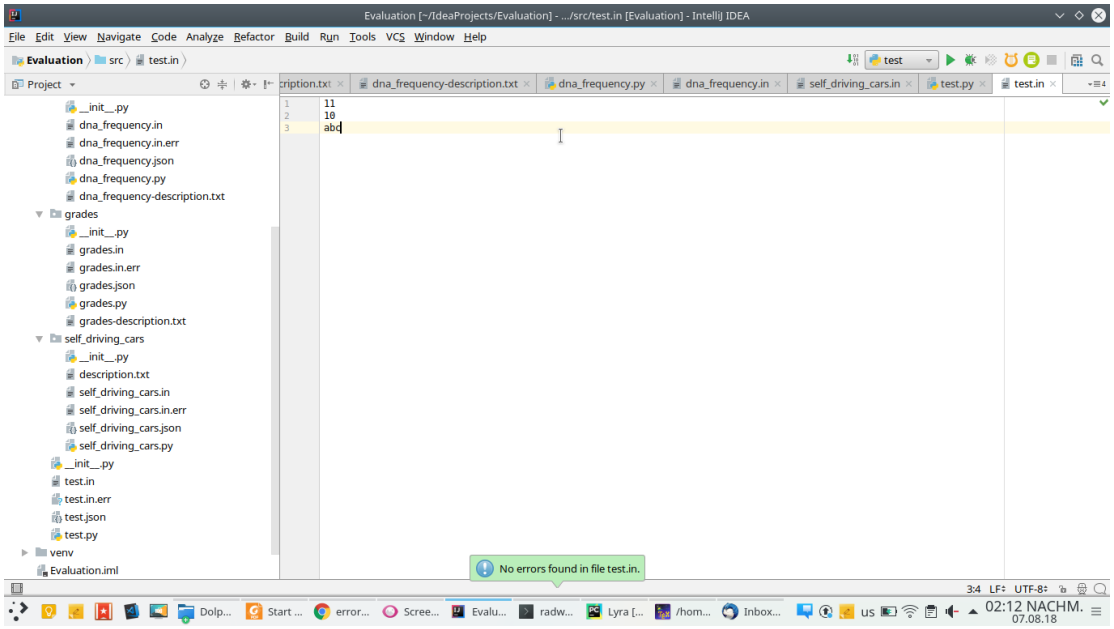
Wrong values are underlined in red in the input file. Hover over the red line to see the error message.



The tool will show a pop-up indicating the first error in the file



When the tool cannot detect any more errors in your input file, it will display a message in a green pop-up like the image below.



Please tell me when you're ready to start the second experiment.

Please answer these questions after finishing the task:

11. **How many minutes did this experiments take?** *

12. **Were you able to fix all the errors in the input file?** *
Mark only one oval.

- Yes
- No

13. **Were you able to run the program without raising en error?** *
Mark only one oval.

- Yes
- No

Please rate how much you agree with the following statements:

14. **I felt frustrated trying to solve this problem** *
Mark only one oval per row.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
Rate					

15. **The error messages that appeared in the input file helped me identify what was wrong with the input** *
Mark only one oval per row.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
Rate					

16. **I found it easy to run the tool and understand its output** *
Mark only one oval per row.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
Rate					

17. **I would prefer to fix errors in the input data using the tool than using program error messages only** *
Mark only one oval per row.

	Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
Rate					

General questions

18. **What did you like about this tool?** *

19. **What suggestions would you have to improve this tool?** *

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.

[8] Antoine Miné et al. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages*, 4(3-4):120–372, 2017.

[9] Lyra project.

[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.