Automated Generation of Data Quality Checks

Master Thesis

Madelin Schumacher

March 2018

Advisors: Dr. Caterina Urban, Alexandra Bugariu

Prof. Dr. Peter Müller
Chair of Programming Methodology
Department of Computer Science, ETH Zürich

Abstract

Data analysis programs are used by computer scientists as well as specialists from other fields to compute useful information based on input data. Even though these programs are correct, unexpected input data can lead to a program failure. Since the program only reports what went wrong in terms of the instructions it tried to execute, users need to analyse the error to find and fix the problematic values in the input file. This task is time consuming or even impossible for specialists without knowledge in programming.

An input value is unexpected by a program if it produces a program error. This happens when a program makes an implicit assumption about an input value that does not hold. For example, if an input is cast to an integer, the program makes the assumption that the input contains numbers only.

In this thesis we present the design of abstract domains that are used for static analysis by abstract interpretation. The result of this analysis is a list of assumptions that need to be fulfilled by the input values. If an input value violates an assumption the program will yield an error.

In addition, an input checking algorithm was designed that uses the assumptions inferred by the static analysis to find problematic input values. The assumptions are matched with the input values and values that violate an assumption are reported to the user together with the expectation of its format.

The static analysis as well as an input checker tool were implemented. The static analysis infers assumptions made by the program about the type and numerical range of an input value and assumptions about relations between two input values. The analysis can be improved to be more precise and additional kinds of assumptions can be added.

We conducted a user study that demonstrates the need and benefit of the input checker tool. The participants were able to resolve problems in the input data fast but the tool was not intuitive enough and we conclude that it would serve much better as a plugin to a text editor.

Acknowledgements

I would like to express my gratitude to Dr. Caterina Urban and Alexandra Bugariu. I am grateful for their guidance and support throughout the entire project. Their comments and insights were enormously helpful to complete my thesis. I want to thank Prof. Dr. Peter Müller for giving me the opportunity to work on this project. My sincere thanks goes to all participants of my user study who have taken their time to provide me with insightful remarks. Finally, I would like to thank my boyfriend, friends and family whose support and encouragement was invaluable.

Contents

1 Introduction

Problem Statement Data analysis programs are developed and used not only by computer scientists but also by biologists, statisticians and specialists from other fields. Even though these programs are correct, unexpected input data can lead to a program failure or to an erroneous output which might be difficult to detect.

Due to a common lack of documentation or because the user is not experienced enough it might not be clear how the input data has to be formatted, what value ranges can be used, what type the input data should have, do values have to be unique and so on. For programs needed in medical environments the input data used for a certain program might not be available to the programmer because of confidentiality issues. This can lead to errors when the program is run by a user because the programmer could not test their software with real input data.

[1] shows that most data errors happen for the following reasons: data is copied from a corrupted input source (e.g. misunderstanding someone at the telephone or working with illegible hand writing), input data that captures measurements of a physical process is wrong because of a flaw in the design or execution, a faulty preprocessing of raw data results in wrong values, or errors occur when merging multiple sources of input data and inconsistencies need to be resolved (e.g. different units, data representation, measurement periods and so on).

Many data analysis programs have to run for many hours to produce a result. It is cumbersome having to restart a program that ran for hours if an error occurred because of faulty input data. Often it is not feasible for the user to check the data by hand because data analysis programs work with a vast amount of input data.

Our goal is to provide the user with an intuitive method to find and fix unexpected input values that would cause the program to run into an error. We assume that the program works correctly so that if the input values are as expected the program does not raise any errors. We do not look for ways to improve the code to deal with unexpected values but we rather want to help a user to find and modify these unexpected input values so that the program can run successfully.

To point out input values to the user that will cause the program to run into an error and to be able to tell them why the error occurred, we first need to find out what expectations a program has of its input data. If the assumptions that a program makes about its input data are known, input values that violate these assumptions can be detected and reported to the user. We want to check the quality of the input data automatically and report the results of these checks to the user.

Static Analysis To find the assumptions that a program makes about its input data we use *Static Program Analysis*. The program is analysed without actually running it but by inspecting its source code. When we analyse the source code of a program we do not only find out what input values cause a problem but also why the program fails. We can then identify what is expected of an input value so that the error does not occur. This knowledge is useful so that we can later report to the user what input values will cause the program to run into an error and what format is expected instead. With a dynamic approach we could find out what values do

and do not cause an error in the program but it is more difficult to decide what the format is the program expects of the input data.

When using static analysis it is not possible to exactly decide whether an input value will cause the program to yield an error or not. But there are techniques that can be used so that static analysis can give an answer to the property we want to decide. The idea is that we do not restrict the answer to be *"yes"* or *"no"* but that the analysis can report *"I do not know"*.

The problem with this approach is that it is unsatisfactory if the answer to the problem is unknown. We therefore aim for a static analysis that is as precise as possible and reports *"yes"* in as many cases as possible while staying sound. This means that the analysis should never report false positives, i.e. the analysis should never return *"yes"* if the answer is untrue and the input value will not yield an error in the program, but attempt to discover as many true positives as possible. With this approach there can be false negatives so that we will miss some input values that would cause an unsuccessful program execution.

During the analysis we want to capture certain properties about the program. In our case these properties are the assumptions that must hold so that no error occurs in the program that is analysed. Before and after every instruction of the program that is analysed, the analysis is in a certain *state*. This state describes the current assumptions for the different program variables and inputs that are used by the program that must hold so that no program error occurs. A *trace* describes a sequence of these states that define a program execution.

The answer that is given by the static analysis should hold for all possible executions of a computer program. So that no matter what branches of a program will be taken in an actual run of the program, the assumptions that are inferred by the analysis must hold for every program path. The set of all possible program executions of a program is not computable so that we cannot analyse all of them separately. What we will do is to approximate the set of all possible executions to get an answer for which we know that it will be true for all executions.

Abstract Interpretation Abstract Interpretation [2] is a theory that can be applied for the purpose of creating a sound approximation. It is a mathematical theory that guides the approximation of the behaviour of a program. Abstract domains are used to define an abstracted state of a program. When implementing an abstract domain one has to define the following structures and functions:

Elements Every abstract domain consists of elements that are used to describe a certain property of a state in a computer program. These elements approximate the current state of a program trace for a certain property. For example, if we want to know about the possible range a program variable can be in at a certain location of a computer program, the abstract domain would consist of two items describing the lower and upper bound of the range, i.e. an interval $[x, y]$.

Order A concrete program state can be approximated with different elements of an abstract domain. For example, if the analysis can conclude that a program

variable holds the value 2 this can be abstracted with the interval $[2, 2]$ but also $[1, 3]$. Because we want an analysis that is as precise as possible we need to know which of two elements is more precise, so we introduce a partial order for the set of elements. The partial order $a \sqsubseteq b$ is reflexive, antisymmetric and transitive. We will say that element a is more precise than b. In our example we know that $[2, 2] \sqsubseteq [1, 3]$.

Join When two program paths are merged because of an if-else-statement or a loop we need to define how to combine the abstract elements on both branches to get an approximated result that is sound and as precise as possible. This is called the join $a \sqcup b$ of abstract elements a and b. The join is used when we know that either one or the other element represents the current program state. The join of an element $a = [1, 2]$ and $b = [8, 10]$ is defined as $[1, 2] \cup [8, 10]$ and would result in $c = [1, 10]$. This result is sound because it contains all elements of both intervals. It is less precise because it also contains elements that are not used by either a or b but most precise because it contains as few additional elements as possible. A result $d = [0, 20]$ would be sound as well but less precise than c. The join is associative and commutative.

Meet The meet $a \sqcap b$ is used when we know that both elements a and b are true for a program state. If we compute the meet of two elements the result will be more or equally precise than each of the elements. The meet can be used if we have different information about states that exist on the same program trace so that we can combine these. The meet of elements $a = [1, 9]$ and $b = [2, 10]$ is defined as $[1, 9] \cap [2, 10]$ and would result in $[2, 9]$. The meet is associative and commutative.

Widening The widening operator is needed to ensure that the analysis converges. If for example a value keeps increasing inside a loop so that the interval of possible values increases at every iteration we need a function that computes a fixed point where the abstract element does not change in the next iteration. If we replace an element with the interval $[-\infty, \infty]$ the interval will not change in the next iteration.

Assign The assign function describes the effect of an assignment $a = e$ on the abstract domain, where a is a variable identifier and e an expression. An assign function will modify the current elements of the abstract domain.

Filter The filter function defines the effect of a condition on the abstract elements. A condition is encountered when a loop or if-else statement is reached where a condition decides what branch is taken. The loop body or the body of an if-else statement can only be reached if the preceding condition was met. The elements of the abstract domain can then be modified to reflect this condition.

Input Checker The result of the analysis is a list of assumptions that need to be fulfilled by the input values because otherwise there will be an error in the program. Each assumption of the list corresponds to an input value that has to met the assumption. We also call this the precondition that has to be satisfied by the input data. This precondition can now be used to find input values that violate an

assumption that is made by the program and to report them to the user. To match an assumption of the list with its corresponding input value we created an input checking algorithm. This algorithm checks if the assumption is met by the input value and reports input values that violate an assumption.

Example Listing 1 shows a running example of a program that is used throughout this thesis. The input data that is used by the program contains information about the weight of a mouse starting from its birth and information about its daily food intake. The goal of the program is to print the weight for a day of interest and the food intake that was recorded for certain days. First an input is read and split by a delimiter indicating multiple input values on one line. These values represent the weight of a mouse per day starting from its birth. Another input, indicating the day of interest, is read and cast to an integer. The list of weights is accessed using the value stored in *day* and the result is printed. Then we iterate for ten times. Inside the loop we read another value that is cast to a float and stored in the variable *other_day*. This value represents the day (including the part of the day as the decimal after the comma) on which the food intake was recorded. If this value is smaller than zero an error is raised. Similarly an error will be raised if the value in *other_day* is smaller than the value in variable *day*. If the value is greater than two we cast the next input to an integer, otherwise we cast it to a float as the food can be a fraction during the first days after birth. The value of $food$ is then printed. Outside of the loop there is another if-else statement. If the value of variable *day* is equal to one, the value at position two of *weight_per_day* is printed, otherwise the value at position three will be printed.

```
1  weight_per_day = input().split(';')
2  day_string = input()
3  day = int(day_string)
4  print(weight_per_day[day])
5  for i in range(10):
6      other_day = float(input())
7      if other_day < 0:
8          raise ValueError
9      if other_day < day:
10         raise ValueError
11     if other_day > 2:
12         food = int(input())
13     else:
14         food = float(input())
15     print(food)
16 if day == 1:
17     print(weight_per_day[2])
18 else:
19     print(weight_per_day[3])
```

Listing 1: Python program used as a running example

In this example we can find the following implicit assumptions that are made by the program about the input data:

- From line 3 we know that the second input must be of type integer because otherwise Python would raise a ValueError.
- From line 4 we can infer that $day < len(weight\_per\_day)$ because otherwise the program would run into an IndexError. Since the variable *day* contains the value from the second input and the first input is stored in variable *weight_per_day*, we can deduce that the value of the second input of the program must be smaller than the number of elements we get when splitting the first input by ';'.
- An implicit assumption is made for the input in line 6 where a value of type float is expected.
- We want to avoid that the program reaches line 8 because we do not want the program to reach the instruction where an explicit error is raised. To avoid reaching this statement we need to invert the condition that leads to the error. Therefore we can infer the assumption that the value stored in *other_day* must be greater or equal to zero.
- Similarly we do not want to reach line 10 which means that the the inverted condition $other\_day \geq day$ must hold. Therefore the input from line 6 must be greater or equal to the value from line 3.
- In line 12 we get the assumption that the input must be of type integer. In line 14 the program assumes that the input is of type float. Because we do not know which branch will be taken in an actual run of the program we cannot decide which assumption has to hold. We can however state that no matter what path will be taken, the value of the input must be at least castable to a float. This solution will be sound because no input will be rejected that would result in a successful program run.
- In line 17 we get the assumption that the first input must have at least three elements. In line 19 the program assumes that the first input must have at least four elements. We can see that the assumption that there must be at least three elements in the list is sound for either branch.

1.1 Related Work

To ensure a successful program execution most approaches check correctness of the program itself by using formal verification ([3], [4]), model checking [5], or other means [6].

In contrast, we assume that the program works correctly for expected input data. Our goal is to ensure successful program execution by checking the input data for correctness. Less research has been done in this area.

CheckCell [7], for example, proposes *data debugging*, an approach to ensure input data correctness. It is a tool to detect data errors in spreadsheets based on the assumption that if one data cell has a big impact on the output compared to the other cells it is either rather important or wrong. CheckCell uses program analysis

to identify output cells and their corresponding inputs. Statistical analysis is used to find out if an input value has a disproportionate influence on the output. The tool gives an impact score to each of the inputs and the user can choose what percentage of input values they want to check. The tool then shows to the user the found values one after another so that they can decide whether the value is wrong.

The approach solely uses the inputs and output of a function without considering the function itself. Input values are compared to each other to find faulty ones. Our approach on the other hand analyzes how the input values are used by the program and what assumptions are made about the input values. Because CheckCell is implemented as an add-in for Microsoft Excel, runtime errors are detected by Microsoft Excel and not the program itself. Our analysis reports values that will produce runtime errors that would otherwise not be detected before running the program.

1.2 Outline

In Section 2 we will describe the static analysis that is used to infer assumptions that are made by a program about its input data. In Section 3 we explain how the input checker works. Details about the implementation of the static analysis and input checker are given in Section 4. An evaluation of the static analysis and input checker is presented in Section 5. The conclusion of this thesis can be found in Section 6.

2 Static Analysis

In this chapter we will presented the static analysis that infers assumptions that a program makes about its input data. In the next section we present more details on how the analyser works. Section 2.2 will show how the analysis deals with non-relational assumptions, i.e. assumptions that do not include relations to other input values. For example in Listing 1 in line 3 we have the assumption that the first input must be an integer which is a non-relational assumption because it refers to the second input value only. In Section 2.3 we present how we improved the analysis to find relational assumptions as well. Those assumptions expect that a certain relation between two input values holds. In Listing 1 we get a relational assumption in line 4 because we relate the value of the second input to the length of elements in the first input.

2.1 Solution

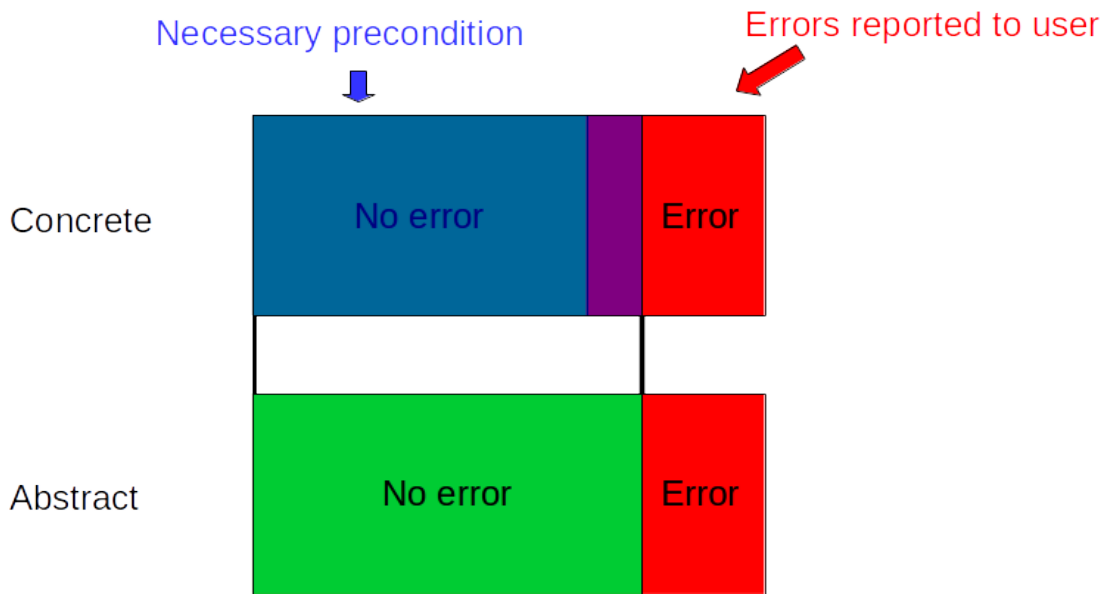


Figure 1: Over-approximation of correct states

Over-Approximation As explained in the introduction the static analysis cannot be exact and has to work with an approximation of the program. Our analysis over-approximates the states that we define as successful. In our case a successful state is one where the program does not yield an error. Figure 1 shows how the over-approximation is done. When we over-approximate the correct states of the program to find the precondition of the program we are sure that all the values that do not fulfil this precondition will yield an error in the program. In other words, we want to find the *necessary precondition* [8]. If this precondition is violated we know that the program will definitely fail. However, because of the over-approxmation we will miss some input values that would produce an error in the program.

Backward Analysis Our analysis starts at the end of the program that is analysed and propagates information up to the start of the program. While doing this the analysis modifies the assumptions as their value changes. The intuition behind doing a backward analysis is that we want to start the analysis at the point where an error can occur. The analysis captures the assumption that must hold so that the error is avoided and propagates that information up to the point where the analysis can infer assumptions about an input value that must hold so that the error does not occur.

To find assumptions that a program makes about its input data, the analysis needs to find out how the inputs are used in the program. Depending on how they are used by the program, certain assumptions are made about the content of the inputs. For example in Listing 1, the input read in line 2 is stored in the program variable *day_string* that is cast to an integer in the next line. We can therefore conclude that the input must be castable to an integer. In a backward analysis, line 3 is encountered before the analysis knows that the program variable *day_string* refers to an input. The analysis first has to capture the assumption that the program variable *day_string* must contain a value that is castable to an integer. The analysis can then infer in line 2 that the assumption that has to hold for *day_string* also has to hold for the input that is read in this line.

Preparation To gain basic information about the types of the variables that are used, an existing type inference algorithm is run before the analysis starts. The type inference algorithm will assign one type to every program variable. The analyser can then use this information to infer an assumption about the type of an input value. The static analysis presented in this thesis can however find more precise type assumptions in some cases. Taking the example from before, the type inference algorithm will conclude that the variable *day_string* is of type string. The assumption analysis however will infer the information that *day_string* must be castable to an integer and therefore obtain more precise assumptions about the input value. Having the type of variables available before the analysis starts also enables the analysis to make assumptions about certain aspects of a program variable. For example, we can add a special program variable $len(data)$ that will gather assumptions about the length of a list called *data*.

To make assumptions about program variables the analysis needs to know what program variables are used by the program. Before the analysis starts the program will be scanned and all variables that are used will be recorded. A default assumption will be assigned to each program variable. These default assumptions indicate that we do not have any specific information about the content and format of a program variable yet. For lists we can add a special length identifier variable with a default assumption for the length of the list. These assumptions are modified by the analyser when more information is found.

2.2 Non-Relational Assumptions

Non-Relational assumptions tell us about the expectations a program has of an input in isolation, without relating it to any other value. Our analysis supports

assumptions about the type of an input value and its range if the value is a numerical one.

As explained in the introduction we need to define an abstract domain to make use of the abstract interpretation theory. Figure 2 shows an overview of the domains that are used by the analysis to infer non-relational assumptions. The arrows indicate what domain elements are used by a domain.

The assumption domain captures assumptions about the type, respectively the range of a program variable. The input assumption domain contains all assumptions the analysis can infer for input values. The input assumption stack domain makes sure that assumptions about inputs in a loop body are captured and the program assumption domain keeps track of all assumptions about program variables and input values.

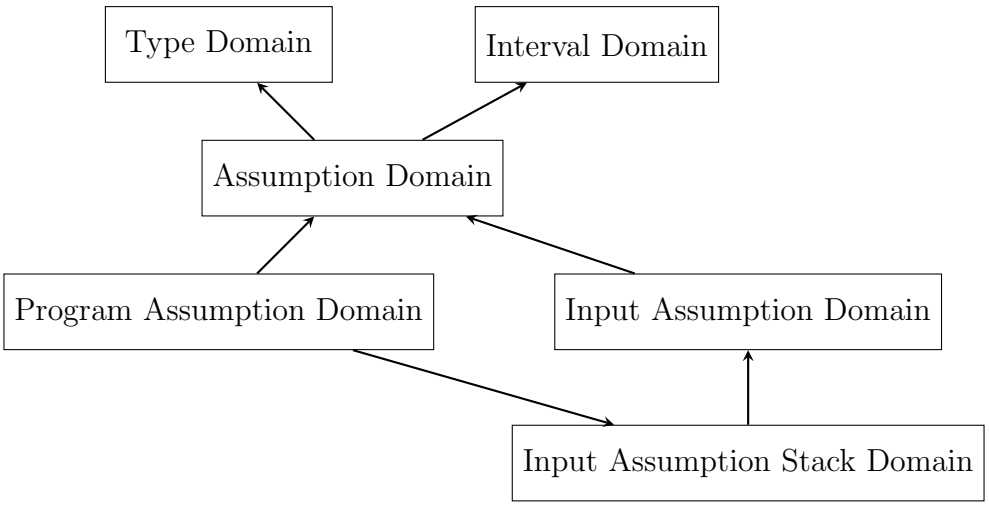


Figure 2: Abstract domains used for the analysis of non-relational assumptions, with arrows indicating other domains whose elements are used by an abstract domain

2.2.1 Type Domain

The type domain defines the structure of the different types a value can have. The lattice of this domain is shown as a Hasse Diagram in Figure 3. The domain is defined as follows:

Elements, Order, Join, Meet The elements and lattice operations $\sqsubseteq, \sqcup, \sqcap$ are defined by the Hasse Diagram. The bottom element is $\bot$ and the top element is *Any*.

Widening $a \bigtriangledown b = a \sqcup b$

Assign For a substitution $a = e$ we compute $type(a) \sqcap type(e)$ where $type(x)$ is the element of the type domain that is associated with the type of the expression.

Filter For all variables that are used in the assume statement we can add the information about the type of the variable that is available because of the type inference algorithm that ran before the analysis started.

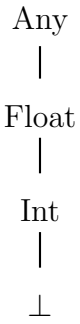


Figure 3: Type Lattice

2.2.2 Interval Domain

The Interval Domain is well-known and further information can be found in [2]. We use this domain to make assumptions about the range of a value if we know that it is of a numerical type.

2.2.3 Assumption Domain

The assumption domain is used by the analysis to represent information about a type and range. It includes elements from the type and interval domain. The assumption domain is defined as follows:

Elements An element a from the assumption domain is defined as $a \in T \times I$ where T is the set of elements from the type domain and I the set of elements from the interval domain. The bottom element is $(\bot, \bot)$ and the top element is $(Any, \top)$.

Order, Join, Meet, Widening, Assign, Filter The operations $\sqsubseteq, \sqcup, \sqcap, \bigtriangledown$ are applied point-wise to each element of the assumption domain.

An element of the assumption domain is for example used in Listing 1 on line 3 where we have an assumption $(Int, [-\infty, \infty])$ for program variable *day_string*.

2.2.4 Input Assumption Domain

At the beginning of the analysis all program variables are initialised with a default element of the assumption domain. These assumptions will be modified by the analyser when more precise information about a program variable is inferred. In the beginning, the analysis knows the program variables that are used by the program but has no information about the structure of the inputs. The input assumption domain is needed to gather knowledge about the structure of the input values and the

assumptions that are made about them. It maintains a list that contains elements from the assumption domain and the order of the elements in the list relates to the order of the inputs in the input file.

Not only the order of the input values is captured but also their multiplicity. If inputs are read inside a loop the assumption that is inferred about these inputs will be the same for all of them. The input assumption domain will contain the information about the number of times an assumption has to hold. The domain is defined as follows:

Elements An element s of the input domain is defined as $s \in \mathbb{N} \times (A \cup S)^n$ where A is the set of elements from the assumption domain, S is the set of elements from the input assumption domain and $\mathbb{N}$ is used to indicate the multiplicity of the assumptions. The assumptions are collected in a list $[a_1, a_2, ..., a_n]$ where a_i is an element from the assumption domain explained before or a nested element from this domain. The order of the assumptions in the list corresponds to the order in which the inputs are read. An element a_i from the assumption domain describes the assumption of one particular input. If a_i is an element from the input assumption domain it indicates that there are assumptions that hold for multiple input values in succession. The bottom element of the domain is $\bot$ and the top element is $\top$.

An element $(1, [\,])$ will be created at the beginning of the analysis and all assumptions about input values found by the analysis will we stored inside the second element that we call the *assumption list*.

We illustrate this using the example from Listing 1. If we take into account the program up to line 6 in isolation we can conclude the following: In line 6 the analysis infers the assumption $(Float, [-\infty, \infty])$ which means that the input that is read on this line must be a float. Because the loop in line 5 will be executed ten times we know that the assumption from line 6 has to hold for ten input values. The input assumption domain element for this case is $(10, [(Float, [-\infty, \infty])])$. This assumption will be added to the assumption list of the element $(1, [\,])$ that was created in the beginning. So we end up with $(1, [(10, [(Float, [-\infty, \infty])])])$ When continuing the analysis backwards the analysis creates the assumption element $(Int, [-\infty, \infty])$ for program variable *day_string* in line 3 to state that the value of the variable must be of type integer. The analysis will add this assumption to the front of the assumption list because this input will be read by the program before the ones that are already in the list:

$$(1, [(Int, [-\infty, \infty]), (10, [(Float, [-\infty, \infty])])])$$

The assumption list of this input assumption domain element indicates that there is an input of type integer that is followed by ten inputs with the assumption that they are of type float.

Order $(N_1, [a_1, a_2, ..., a_n]) \sqsubseteq (N_2, [b_1, b_2, ..., b_m]) = (n = m \wedge \forall i \in [1, n] : a_i \sqsubseteq b_i)$

Join For the join we have a case distinction, depending on whether we do the join because of a loop ($IsLoop$) or an if-else statement ($IsCase$).

- $IsLoop \wedge length([A]) > length([B]) \Rightarrow (N_1, [A]) \sqcup (N_2, [B]) = (N_1, [A])$
- $IsLoop \wedge length([A]) \leq length([B]) \Rightarrow (N_1, [A]) \sqcup (N_2, [B]) = (N_2, [B])$
- $IsCase \wedge length([a_1, ..., a_n]) = length([b_1, ..., b_m]) \wedge N_1 = N_2$
  $\Rightarrow (N_1, [a_1, ..., a_n]) \sqcup (N_2, [b_1, ..., b_m]) = (N_1, [a_1 \sqcup b_1, ..., a_n \sqcup b_n])$

  The last case is used to join assumption lists from two branches of an if-else statement that both have the same iteration number and contain the same number of assumptions. The join of the lists is done point-wise. In our running example we have a case distinction in line 11. In both branches we encounter one input. For one branch we get the input assumption $(1, (Int, [-\infty, \infty]))$, for the other one we get $(1, (Float, [-\infty, \infty]))$. We can then join these two branches by joining the elements of the assumption lists:

$$
\begin{aligned}
&(1, (Int, [-\infty, \infty])) \sqcup (1, (Float, [-\infty, \infty])) \\
&= (1, (Int, [-\infty, \infty]) \sqcup (Float, [-\infty, \infty])) \\
&= (1, (Int \sqcup Float, [-\infty, \infty] \sqcup [-\infty, \infty])) \\
&= (1, (Float, [-\infty, \infty]))
\end{aligned}
$$

In cases when we join branches where the if and else branch have a different number of input values, we have to be careful so that the analysis stays sound. The analysis needs to infer an input assumption domain element that covers both branches. An example for this case can be found in Listing 2. The program reads an input in line 1 and depending on that value, one more input will be read if the branch condition is true and otherwise three more float values will be read. In the end another value of type integer is read. The program therefore uses a different number of input values depending on what branch is taken.

We designed an algorithm that computes the join of two branches with different number of inputs so that the analysis will get a result that is sound and precise as well. The main idea is that we take the input assumption list of all possible program paths and join them. The algorithm joins the elements of the assumption lists from left to right in a recursive manner and stops if one or both of the lists have no more elements that can be joined. To do the join using nested assumptions, the structure of the input assumption domain element can be flattened.

To explain the algorithm in detail we use A as the set of assumption domain elements and the set S for elements of the input assumption domain. The algorithm computes $(1, [x_1, x_2, ..., x_n]) \sqcup (1, [y_1, y_2, ..., y_m])$ for $x_i, y_i \in A \cup S$ as follows:

```
1 cond = input()
2 if cond == "True":
3     x = input()
4 else:
5     for i in range(3):
6         y = float(input())
7 z = int(input())
```

Listing 2: Program with a different number of inputs in if and else branch

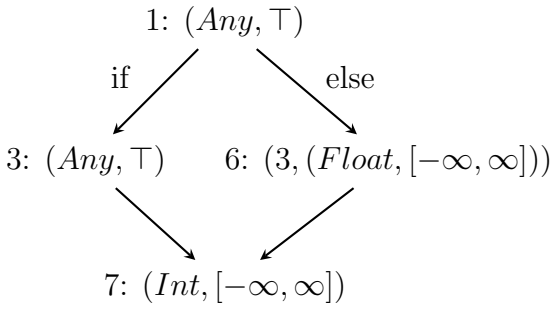


Figure 4: Input assumptions of program on the left side with corresponding line numbers

1. $(1, [x_1, x_2, ..., x_n]) \sqcup (1, [y_1, y_2, ..., y_m]) = (1, [x_1, x_2, ..., x_n] \sqcup [y_1, y_2, ..., z_m])$

2. For case $x_1, y_1 \in A$:

$$[x_1, x_2, ..., x_n] \sqcup [y_1, y_2, ..., y_m] = [x_1 \sqcup y_1] \cdot ([x_2, ..., x_n] \sqcup [y_2, ..., y_m])$$

3. For case $x_1 \in A$ and $y_1 = (k, [z_1, z_2, ..., z_p]) \in S$:

$$\begin{aligned} &[x_1, x_2, ..., x_n] \sqcup [(k, [z_1, z_2, ..., z_p]), y_2, ..., y_n] \\ &= [x_1, x_2, ..., x_n] \sqcup [z_1, z_2, ..., z_p, (k-1, [z_1, z_2, ..., z_p]), y_2, ...y_n] \end{aligned}$$

4. For case $x_1 = (j, [v_1, v_2, ..., v_l]) \in S$ and $y_1 = (k, [z_1, z_2, ..., z_p]) \in S$ and $j = k$:

$$\begin{aligned} &[(j, [v_1, v_2, ..., v_l]), x_2, ..., x_n] \sqcup [(k, [z_1, z_2, ..., z_p]), y_2, ..., y_m] \\ &= [(j, ([v_1, v_2, ..., v_l] \sqcup [z_1, z_2, ..., z_p])] \cdot ([x_2, ..., x_n] \sqcup [y_2, ..., y_m]) \end{aligned}$$

5. For case $x_1 = (j, [v_1, v_2, ..., v_l]) \in S$ and $y_1 = (k, [z_1, z_2, ..., z_p]) \in S$ and $j < k$:

$$\begin{aligned} &[(j, [v_1, v_2, ..., v_l]), x_2, ..., x_n] \sqcup [(k, [z_1, z_2, ..., z_p]), y_2, ..., y_m] \\ &= [(j, [v_1, v_2, ..., v_l]) \sqcup (j, [z_1, z_2, ..., z_p])] \\ &\quad \cdot ([x_2, ..., x_n] \sqcup [(k-j, [z_1, z_2, ..., z_p]), y_2, ..., y_m]) \end{aligned}$$

6. In the case $[\,] \sqcup [y_1, y_2, ..., y_m]$ we end the algorithm and discard the remaining assumptions $[y_1, y_2, ..., y_m]$

All symmetrical cases are similar.

This algorithm can only be used if one of the branches that is joined contains the main execution path of the program. In the main path we have all the information about the inputs that follow after the current point that is currently analysed. We illustrate how this join of branches with different number

of inputs works and why we can only do this if it involves the main path of the program:

Figure 4 visualises what assumptions are found at the program points in the different paths of the program left of it. For Listing 2 the analysis would infer the following input assumption domain element for the program path where the branch condition is true:

$$(1, [(Any, \top), (Any, \top), (Int, [-\infty, \infty])])$$

The first assumption of the list corresponds to the input found in line 1, the second assumption to the input that is read inside the if branch in line 3 and the third input is from line 7 where the analysis infers that the input must be of type integer.

In a similar manner we get the following input assumptions for the case where the else branch is taken by the program:

$$(1, [(Any, \top), (3, (Float, [-\infty, \infty])), (Int, [-\infty, \infty])])$$

The second element of the assumption list corresponds to the three values that we read in line 6. In this branch the program reads a total of five values whereas there are only three in the other branch.

The previously explained join algorithm would work in this example as follows[1]:

$$(1, [(Any), (Any), (Int)]) \sqcup (1, [(Any), (3, (Float)), (Int)])$$

This corresponds to case 1 of the algorithm and we get:

$$(1, [(Any), (Any), (Int)] \sqcup [(Any), (3, (Float)), (Int)])$$

To join the two assumption lists, case 2 of the algorithm applies:

$$\begin{aligned} & [(Any) \sqcup (Any)] \cdot ([(Any), (Int)] \sqcup [(3, (Float)), (Int)]) \\ = \; & [(Any)] \cdot ([(Any), (Int)] \sqcup [(3, (Float)), (Int)]) \end{aligned}$$

In the next step case 3 is used:

$$[(Any)] \cdot ([(Any), (Int)] \sqcup [(Float), (2, (Float)), (Int)])$$

Now case 1 can be used again and we get:

$$\begin{aligned} & [(Any)] \cdot [(Any) \sqcup (Float)] \cdot ([(Int)] \sqcup [(2, (Float)), (Int)]) \\ = \; & [(Any)] \cdot [(Any)] \cdot ([(Int)] \sqcup [(2, (Float)), (Int)]) \end{aligned}$$

Repeating the last two steps we get:

$$\begin{aligned} & [(Any)] \cdot [(Any)] \cdot ([(Int)] \sqcup [(2, (Float)), (Int)]) \\ = \; & [(Any)] \cdot [(Any)] \cdot ([(Int)] \sqcup [(Float), (1, (Float)), (Int)]) \\ = \; & [(Any)] \cdot [(Any)] \cdot [(Int) \sqcup (Float)] \cdot ([\,] \sqcup [(1, (Float)), (Int)]) \\ = \; & [(Any)] \cdot [(Any)] \cdot [(Float)] \cdot ([\,] \sqcup [(1, (Float)), (Int)]) \end{aligned}$$

[1]The range assumptions for this example are omitted for improved readability.

With this we reach case 6 and the final solution for the join is:

$$(1, [(Any), (Any), (Float)])$$

This solution states that we expect three input values where the last one must of type float. This result is sound because the analysis will never reject inputs that would not raise an error in the program.

To see why this algorithm only works for joins that involve the main path we can look at the example of Listing 3. For this example the program from the example before was taken and put inside a loop that executes two times. The instructions that read an input are now nested in an if-else statement that is more than one branch away from the main path.

```
1 for i in range(2):
2     cond = input()
3     if cond == "True":
4         x = input()
5     else:
6         for i in range(3):
7             y = float(input())
8     z = int(input())
```

Listing 3: Program with a different amount of input in a deeply nested if and else branch

```
False
1
2
3
4
True
11
12
```

Listing 4: Input example for the program on the left for a successful run

For the previous program we got the final assumption $(1, [(Any), (Any), (Float)])$. When we now analyse the loop body of the program in Listing 3, using the join algorithm as before, we would end up with the same assumption in line 2. When the analysis now encounters the loop header in the next step, the analysis can infer that the loop executes two times and will therefore modify the input assumption domain element to indicate that the assumptions hold two times: $(1, [(2, [(Any), (Any), (Float)])])$. This solution is not sound because the example input in Listing 4 would be rejected by this precondition even though the program would run successfully. The assumptions and inputs are matched as shown below and the input value *True* will be rejected because an input of type float is expected.

False $\leftrightarrow$ (Any)
1 $\leftrightarrow$ (Any)
2 $\leftrightarrow$ (Float)
3 $\leftrightarrow$ (Any)
4 $\leftrightarrow$ (Any)
True $\leftrightarrow$ (Float)
11
12

As mentioned before the main idea of the join algorithm is to join the input assumption lists of all possible program paths. The program in Listing 3 has a total of four different paths that can be taken. At each loop iteration a different branch can be taken depending on whether the condition $cond == "True"$ is true. We can infer the following four input assumption lists:

1. $(1, [(Any), (Any), (Int), (Any), (Any), (Int)])$
2. $(1, [(Any), (Any), (Int), (Any), (3, [(Float)]), (Int)])$
3. $(1, [(Any), (3, [(Float)]), (Int), (Any), (Any), (Int)])$
4. $(1, [(Any), (3, [(Float)]), (Int), (Any), (3, [(Float)]), (Int)])$

If we apply the join algorithm to these four assumption lists the result is

$$(1, [(Any), (Any), (Float), (Any), (Any)])$$

which is different to $(1, [(2, [(Any), (Any), (Float)])])$ which is the result of the analysis when the join algorithm is used for the if-else statement in line 3 of the program in Listing 3.

The reason why the join algorithm does not work in this case is that the join algorithm discards input assumptions when inspecting the if-else statement as mentioned in step 6 of the join algorithm. If we discard input assumptions at some point during the analysis we need to discard all input assumptions that were already gathered by the analysis as well. Since we gather all assumptions relative to each other we lose the information of how they relate to each other as soon as we discard one input assumption.

For example, when the analysis gathers assumptions $[a_1, a_2, a_3, a_4]$ for the inputs $[l_1, l_2, l_3, l_4]$ we will relate the assumption at a position i in the list with an input at the same position. If for some reason we have to discard assumption a_3, we cannot keep the list $[a_1, a_2, a_4]$ because we would wrongly match assumption a_4 with input l_3.

For this reason the join algorithm can only be used to join branches where the main program path is involved. When joining branches of the main path, all currently gathered input assumptions are involved and if assumptions have to be discarded because of step 6 of the join algorithm, there will be no holes in the list since the whole tail of the input assumption list will be discarded.

Meet The meet of two input assumption domain elements is not used by the analysis.

Widening Widening is defined as the join of two input assumption domain elements.

Assign, Filter The assign and filter operations are not used by the input assumption domain.

2.2.5 Input Assumption Stack Domain

When two program branches are merged, the analysis joins the assumptions of these branches to create one assumption that covers both branches. This assumption must make sure that no matter what branch will be taken, every input value that violates the assumption forces the program to yield an error.

A program is for example split into two branches when a loop is encountered. In one branch we have all the assumptions that we gathered before we encountered the loop. The other branch contains the assumptions that are found inside the loop body. All assumptions that were found inside the loop that are more precise than the assumptions outside the loop have to be discarded when the join is performed. The analyser can only keep assumptions that hold in all program paths. The assumption of the loop body is therefore lost when performing a join. For example in Listing 1 we would lose all information that we found inside the loop if we simply perform the join of the loop using the abstract domains we presented so far.

This behaviour is unsatisfactory, especially when input values are read inside a loop. If the analysis is not able to include the information that an input value is read inside a loop we have to discard all information we have about inputs so far because we keep them relative to each other in the input assumption list. To deal with this problem we created an abstract domain that allows us to analyse the body of a loop in isolation so that we can keep the knowledge of input values that are read inside a loop.

In the example of Listing 1 we know that the loop is going to be executed exactly ten times. This information can be incorporated to create a more precise analysis. When a loop header is reached for which we know the number of times it executes we can take the assumptions found inside the current loop and add it as an element from the input assumption domain with the information that the assumptions holds for a certain number of consecutive input values.

Since we have separate abstract domains for the assumptions of input values and program variables, the usage of the input assumption stack domain will only make the analysis of input assumptions more precise but not the analysis of assumptions about program variables. If the body of a loop makes assumptions about a program variable that are more precise than the assumptions that were gathered before the loop is encountered, the analyis will join these assumptions and the result will be less precise than the assumptions that were encountered inside the loop.

The input assumption stack domain is defined as follows:

Elements The elements of the input assumption stack domain K consist of stacked elements from the input assumption domain: $s_1 \mid s_2 \mid ... \mid s_n$ where s_n

denotes the top of the stack and $s_i \in S$ where S is the set of elements from the input assumption domain.

Order, Join, Meet The operation $k_1 \star k_2$ for $k_i \in K$ and $\star \in \{\sqsubseteq, \sqcup, \sqcap\}$ is defined by applying the operation for each stack layer:

$$(s_1 \mid s_2 \mid ... \mid s_n) \star (t_1 \mid t_2 \mid ... \mid t_n) = s_1 \star t_1 \mid s_2 \star t_2 \mid ... \mid s_n \star t_n$$
$$for\ s, t \in S.$$

Widening $k_1 \bigtriangledown k_2 = k_1 \sqcup k_2$

Assign, Filter The assign and filter operator is not used.

Additionally we define two functions *push* and *pop* that describe the behaviour of adding a new layer on top of the stack, respectively removing the top layer:

Push A new element $(1, [\,])$ from the input assumption domain is added on top of the stack. This is done every time the analysis enters a loop or if-else statement.

Pop The pop operation is different depending whether we pop an element because we exit a loop body or an if-else statement. The pop operation involves the top two elements of the stack. The pop operation for an if-else statement is defined as follows:

$$pop((1, [a_1, ..., a_n]) \mid (1, [b_1, ..., b_m])) = (1, [b_1, ..., b_m, a_1, ..., a_n])$$

This means that we add the assumptions we found inside the statement in front of the assumptions from the underlying layer. The assumptions from that second layer are the ones the analysis found before, so assumptions of inputs that are read later in the direction of the program execution.

For example in Listing 1 we can infer in line 14 a stack

$$(1, [\,]) \mid (1, [\,]) \mid (1, [(Float, [-\infty, \infty])])$$

We have three levels because we are inside a loop and if-else statement. The top element describes that there is an input of type float. When the pop operation is applied to that stack we get the following result:

$$(1, [\,]) \mid (1, [(Float, [-\infty, \infty])])$$

If we already had an assumption (Any, T) on a lower layer of the stack because the analysis already encountered an input the example would work as follows:

$$pop((1, [\,]) \mid (1, [(Any, T)]) \mid (1, [(Float, [-\infty, \infty])]))$$
$$= (1, [\,]) \mid (1, [(Float, [-\infty, \infty]), (Any, T)])$$

When exiting a loop statement we want to keep input assumptions from inside the loop if we are sure how many times the loop body gets executed. If we

have that information (like for example in Listing 1 we know that the loop executes ten times) we will put the list of input assumptions that we found inside the loop in front of the assumptions of the underlying layer and add the number of times that they are iterated:

$$pop((1, [a_1, ..., a_n]) \mid (1, [b_1, ..., b_m])) = (1, [(k, [b_1, ..., b_m]), a_1, ..., a_n])$$
where k is the number of times the loop is executed.

Using this operation we need to make sure that the analysis does not keep on adding the same element to the input assumption list at each loop iteration. In the pop operation the analysis checks if the program point that is analysed was already encountered before. If this is the case the front element of the underlying stack layer will first be removed and the new element is added in the front.

If the analysis cannot determine a constant number of times that the loop is executed, the current list of input assumptions is emptied because we cannot have holes in the list.

2.2.6 Program Assumption Domain

The program assumption domain contains the assumptions associated for each program variable and the current list of input assumptions. It is defined as follows:

Elements An element of the program assumption domain P is defined as (f, k) where f is a mapping from program variables to elements from the assumption domain and k is an element from the input assumption stack domain. The mapping f keeps track of the current non-relational assumption for each program variable and the input assumption stack element k contains the assumptions for the input values.

Order, Join, Meet, Widening An operation $(f_1, k_1) \star (f_2, k_2)$ for $(f_i, k_i) \in P$ and $\star \in \{\sqsubseteq, \sqcup, \sqcap, \bigtriangledown\}$ is computed by applying the operation to both elements point-wise: $(f_1 \star f_2, k_1 \star k_2)$ where $f_1 \star f_2 = f_1(v) \star f_2(v)$ for $v \in V$.

Assign For an assignment of the form $a = e$ where $e \neq input()$, the assign operator defined for each element in the map f is applied. If the assignment is of the form $a = input()$ the element $f(a)$ is added to the list of assumptions in the top layer input assumption element of the input assumption stack element. The assumption $f(a)$ will be reset to the default element.

Filter Similarly as for the assign operator the filter operator for the program assumption domain is computed by applying the filter operator of each element in the map p.

2.2.7 Example

A complete analysis of the example in listing 1 can be found in Section 2.4.

2.3 Relational Assumptions

A relational assumption is used to describe a constraint between two input values. In the example of Listing 1 a relational constraint can be inferred in line 4 where the implicit assumption is made that $day < length(weight\_per\_day)$. Otherwise the program would yield an IndexError.

The design of the analysis presented before was adapted to include relational assumptions. Additionally we added support for assumptions about lists in a program and assumptions of values that are on the same line of an input file separated by a delimiter. Figure 5 presents the abstract domains that are used for the analysis with relational assumptions.

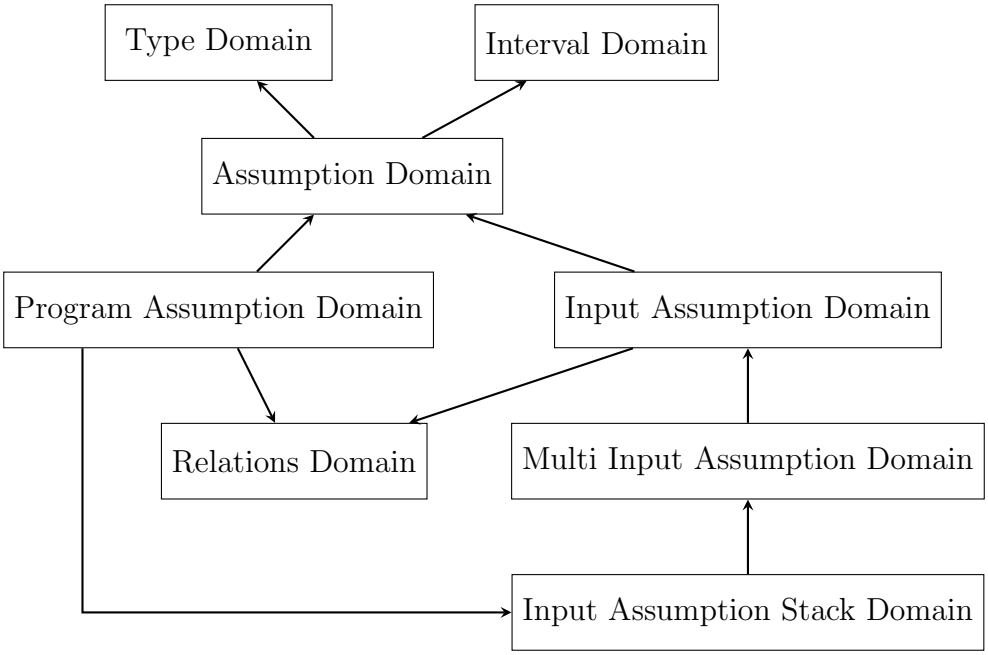


Figure 5: Abstract domains used for the analysis that includes relational assumptions, with arrows indicating other domains whose elements are used by an abstract domain

The definitions for the type domain, interval domain and assumption domain are unchanged and can be found in Section 2.2.1 through Section 2.2.3. The input assumption domain, as well as the program assumption domain, will now contain an additional element to include relational assumptions that are made about program variables and input values. The ability to define the multiplicity of an assumption list was moved to its own domain, the *Multi Input Assumption Domain*. This domain will also contain information about inputs that contain multiple values that are separated by a delimiter.

The adapted and newly created abstract domains are presented next.

2.3.1 Relations Domain

To represent relations between program variables and input values we created a domain that supports relations of the form $\pm x \pm y \leq c$ where x and y are program

variables or input identifiers and c is a constant value. It supports a subset of features from the octagon domain [9]. A special variable var_0 that holds the value zero can be used instead of a program variable to represent a constant value that does not relate to an other variable. This can for example happen if a relational assumptions between two variables is captured and one of them will be substituted with a constant value.

The equality of two relations $r_1 : \pm x_1 \pm y_1 \leq c_1$ and $r_2 : \pm x_2 \pm y_2 \leq c_2$ is defined as:

$$c_1 = c_2 \wedge (x_1 = x_2 \wedge y_1 = y_2 \vee x_1 = y_2 \wedge y_1 = x_2) \Rightarrow r_1 = r_2$$

The domain is defined as follows:

Elements An element of the relations domain is a set of relations $R = \{r_1, ..., r_n\}$ where r is of the form $\pm x \pm y \leq c$.

Order $R_1 \sqsubseteq R_2 = R_1 \subset R_2$

Join $R_1 \sqcup R_2 = R_1 \cap R_2$

Meet $R_1 \sqcap R_2 = R_1 \cup R_2$

Widening $R_1 \bigtriangledown R_2 = R_1 \sqcup R_2$

Assign The domain supports assignments of the form $a = e$ where a is a variable identifier and e is an expression. The analysis will iterate through the set of relations and compute the substitution. If $e \neq input()$ the analysis attempts to convert e into the form $\pm x + c$. If it succeeds the relation will be updated. Otherwise it is removed from the set. In the case $a = input()$ we substitute all occurrences of a in the relations with an input identifier. This input identifier is defined as $id <INT>$ where $<INT>$ is substituted with the program point number for the instruction $a = input()$.

Filter If a branch assumes a statement of the form $a \star b$ where a and b are program variables or constants and $\star \in \{<, \leq, \geq, >\}$ it will be added to the set of relations.

2.3.2 Input Assumption Domain

The input assumption domain stores all assumptions about inputs. It is defined as follows:

Elements An element of the input assumption domain is defined as (a, R, i) where $a \in A$ is an element from the assumption domain, R is a set of relations from the relations domain and i is an input identifier that refers to the program point where the input was read in the program. The element a represents the non-relational assumptions and R the relational assumptions about an input with the identifier i. The input identifier will be of the form $id <INT>$ where $<INT>$ will be substituted with the program point the input was read.

For example in Listing 1 we have the non-relational assumption $(Float, [0, \infty])$ for the input of line 6. Because we do not want to reach line 10 the relational assumption $other\_day \geq day$ must hold. These informations are represented as an input assumption element $((Float, [0, \infty]), \{id6 \geq day\}, id6)$.

Order $(a_1, R_1, i_1) \sqsubseteq (a_2, R_2, i_2) = (a_1 \sqsubseteq a_2 \wedge R_1 \sqsubseteq R_2 \wedge i_1 = i_2)$

Join $(a_1, R_1, i_1) \sqcup (a_2, R_2, i_2) = (a_1 \sqcup a_2, R_1 \sqcup R_2, min(i_1, i_2))$

Meet $(a_1, R_1, i_1) \sqcap (a_2, R_2, i_2) = (a_1 \sqcap a_2, R_1 \sqcap R_2, min(i_1, i_2))$

Widening $(a_1, R_1, i_1) \bigtriangledown (a_2, R_2, i_2) = (a_1, R_1, i_1) \sqcup (a_2, R_2, i_2)$

Assign The assign operator of the input assumption domain will apply the assign operator for the assumption domain element and the element from the relations domain.

Filter The filter operator is not used by the input assumption domain.

2.3.3 Multi Input Assumption Domain

Similarly to the input assumption domain for the analysis with non-relational assumptions only, we will use the multi input assumption domain for input assumptions that hold for multiple values because of a loop. This domain is also used to indicate that an input consists of multiple values that are separated using a delimiter. The elements and operations are defined as follows:

Elements An element m of the multi input assumption domain M is defined as $m \in E \times (A \cup M)^n \times (D \cup \varepsilon)$ where $(A \cup M)^n$ is a list of elements from the input assumption domain or multi input assumption domain and represents a list of assumptions about the input data. E is the set of expression of the form $\pm x + c$ with x being a program variable or input identifier and c is a constant. For x the special variable var_0 can be used to indicate the value zero. This expression indicates the number of times the assumptions in $(A \cup M)^n$ have to hold for the inputs. This is an adapted version of the set $\mathbb{N}$ used in the input assumption domain for non-relational assumptions in Section 2.2.4. Using an expression instead of a constant we can now relate the number of times assumptions have to hold to an other input variable. An element $d \in D$ is a string that is used as delimiter. The special symbol ε is used to indicate that no delimiter is used. If $d = \varepsilon$, each assumption of the list holds for a different input value. If $d \neq \varepsilon$, an input will be treated as a list of values separated by delimiter d. The expression e will then indicate the minimum number of inputs that are expected.

As shown in Section 2.3.2 we get the input assumption element $((Float, [0, \infty]), \{id6 \geq day\}, id6)$ for line 6 in Listing 1. Because of the loop this assumption has to hold ten times. This is represented as a multi input assumption element $(var_0 + 10, [((Float, [0, \infty]), \{id6 \geq day\}, id6)], \varepsilon)$.

For the input on line 1 we know that it consists of a list of values separated by a semicolon. The number of elements must be at least three as explained in the

beginning of Section 2. This will be represented as the multi input assumption element $(var_0+3, [(Any, \top)], ;)$. If the delimiter is not ε the expression var_0+3 represents the number of minimum elements we expect to find. The list of assumptions will only contain one input assumption element which means that the assumption has to hold for every element in the list.

Order $(e_1, [a_1, ..., a_n], d_1) \sqsubseteq (e_2, [b_1, ..., b_m], d_2) =$
$(n = m) \wedge (d_1 = d_2) \wedge \forall i \in [1, n] : type(a_i) = type(b_i) \wedge a_i \sqsubseteq b_1$
where $type(x)$ indicates if x is an element from the input assumption domain or multi input assumption domain.

Join The join of two multi input assumption elements works similar to the join of two input assumption elements from the non-relational analysis in Section 2.3.2. The analysis again distinguishes between the join of two branches from a loop or if-else statement. For the join of two elements from a loop we will make use of the smallest input identifier id that is used in the list of assumptions. This will provide us with the information which element was created when the loop body was analysed and which element comes from the branch that includes the path after the loop body. The join of the elements is done as follows:

- $id_1 = id_2 \wedge length([A]) > length([B])$
  $\Rightarrow (e_1, A, d_1) \sqcup (e_2, B, d_2) = (e_1, A, d_1)$
- $id_1 < id_2$
  $\Rightarrow (e_1, [a_1, ..., a_n], d_1) \sqcup (e_2, [b_1, ..., b_m], d_2) = (e_1, [a_1, b_1, ..., b_m], d_1)$
- $id_1 \geq id_2$
  $\Rightarrow (e_1, [a_1, ..., a_n], d_1) \sqcup (e_2, [b_1, ..., b_m], d_2) = (e_1, [b_1, a_1, ..., a_n], d_1)$

The smaller identifier represents the smaller program point which is the one of the loop header. After the loop header has been analysed the first element of the assumptions list contains the assumptions about inputs that were found inside the loop body. The reason for this will be explained in Section 2.3.4. What we want to a achieve with this join is that the assumptions that were found inside the loop are connected to the assumptions that follow after the loop.

To join two branches of an if-else statement we do the following:

- $length([a_1, ..., a_n]) = length([b_1, ..., b_m]) \wedge e_1 = e_2 \wedge d_1 = d_2$
  $\wedge \forall i \in [1, n] : type(a_i) = type(b_i)$
  $\Rightarrow (e_1, [a_1, ..., a_n], d_1) \sqcup (e_2, [b_1, ..., b_m], d_2) = (e_1, [a_1 \sqcup b_1, ..., a_n \sqcup b_n], d_1)$
  where $type(x)$ indicates if x is an element from the input assumption domain or multi input assumption domain.

In all other cases we run into the same problems as mentioned in Section 2.3.2 and have to discard all currently gathered assumptions, except if the join involves the main branch so that we can make use of the join algorithm explained in Section 2.3.2. The algorithm can be adapted to work for multi input assumption elements. The necessary modifications are that expressions of the

form $\pm x + c$ instead of constants are used and that we only want to join branches that use the same delimiter.

Meet The meet of two multi input assumption elements is not used by the analysis.

Widening $m_1 \bigtriangledown m_2 = m_1 \sqcup m_2$

Assign The assign operator is applied to every element in the list of assumptions.

Filter The filter operator is not used.

2.3.4 Input Assumption Stack Domain

The input assumption stack domain is defined similar to the one used for non-relational assumptions as explained in Section 2.2.5. Instead of using input assumption elements as stack elements we use elements of the multi input assumption domain. The default element that is pushed on top of the stack is $(var_0 + 1, \{\}, \varepsilon)$. The definitions of Section 2.2.5 can be reused with little modification:

Elements The elements of the input assumption stack domain K consist of stacked elements from the multi input assumption domain: $m_1 \mid m_2 \mid ... \mid m_n$ where m_n denotes the top of the stack and $m_i \in M$ where M is the set of elements from the multi input assumption domain.

Order, Join, Meet The operation $k_1 \star k_2$ for $k_i \in K$ and $\star \in \{\sqsubseteq, \sqcup, \sqcap\}$ is defined by applying the operation for each stack layer:

$$(m_1 \mid m_2 \mid ... \mid m_n) \star (t_1 \mid t_2 \mid ... \mid t_n) = m_1 \star t_1 \mid m_2 \star t_2 \mid ... \mid m_n \star t_n$$
$$for\ m, t \in M.$$

Widening $k_1 \bigtriangledown k_2 = k_1 \sqcup k_2$

Assign The assign operator of each element on the input assumption stack will be applied.

Filter The filter operator is not used.

2.3.5 Program Assumption Domain

The program assumption domain contains the assumptions that are associated with the program variables and it contains the input assumptions. It also keeps track of the current relational assumptions. It is defined as follows:

Elements An element p of the program assumption domain P is defined as (f, k, r) where f is a mapping from program variables to elements from the assumption domain, k is an element of the input assumption stack domain and r is an element from the relations domain that contains a set of relational assumptions.

Order, Join, Meet, Widening An operation $(f_1, k_1, r_1) \star (f_2, k_2, r_2)$ for $(f_i, k_i, r_i) \in P$ and $\star \in \{\sqsubseteq, \sqcup, \sqcap, \bigtriangledown\}$ is computed by applying the operation to every element in the map and to the elements of the input assumption stack domain and relations domain: $(f_1, k_1, r_1) \star (f_2, k_2, r_2) = (f_1(v) \star f_2(v), k_1 \star k_2, r_1 \star r_2)$ where v is a program variable used by the program that is analysed.

Assign For an assignment of the form $x = e$ for $e \neq input()$ the analysis applies the assign operator defined for each element $f(v)$ where v are the program variables. Furthermore the assign operator for the input assumption stack domain element and relations domain element are applied.

For an assignment of the form $x = input()$ the assign operators for the input assumption stack domain element and the relations domain element are applied first. Then a new input assumption domain element (a, R, i) will be created. Element a represents the current non-relational assumption we have for x, which is $f(x)$. For R we extract all relations from the relations domain element that were substituted for i which is the program point of the instruction $x = input()$. The element (a, R, i) is added at the front of the list of assumptions in the top layer of the input assumption stack element. The element $f(x)$ will then be reset to the default element.

Filter The filter operation for every element in the map f is applied and the filter operation of the relation domain element r is executed.

2.4 Example

We now present the complete analysis of the running example from Listing 1. A type inference algorithm is run before the analysis starts which results in the program shown in Listing 5

```
1  weight_per_day: List[str] = input().split(';')
2  day_string: str = input()
3  day: int = int(day_string)
4  print(weight_per_day[day])
5  for i in range(10):
6      other_day: float = float(input())
7      if other_day < 0:
8          raise ValueError
9      if other_day < day:
10         raise ValueError
11     if other_day > 2:
12         food: int = int(input())
13     else:
14         food: float = float(input())
15     print(food)
16 if day == 1:
17     print(weight_per_day[2])
18 else:
19     print(weight_per_day[3])
```

Listing 5: Python program used as a running example

1. The analysis will be initialized with an element of the program assumption domain. It includes a map containing all program variables and a special program variable for the length of the list that is used in the example. A default assumption domain element will be mapped to each of these variables. For the length identifier the default assumption is $(Int, [0, \infty])$ because we know that the length of a list must always be an integer greater or equal to zero.

$$\begin{aligned} weight\_per\_day &\rightarrow (Any, \top) \\ len(weight\_per\_day) &\rightarrow (Int, [0, \infty]) \\ day\_string &\rightarrow (Any, \top) \\ day &\rightarrow (Any, \top) \\ other\_day &\rightarrow (Any, \top) \\ food &\rightarrow (Any, \top) \end{aligned}$$

The input assumption stack is initialized with one layer that contains the default multi input assumption element

$$inputs : (1, [\,], \varepsilon)$$

The set of relations is initialized with the empty set

$$relations : \{\,\}$$

2. At the end of the program we enter the else branch of a case distinction. An new layer is pushed onto the input assumption stack:

$$inputs : push((1, [\,], \varepsilon)) = (1, [\,], \varepsilon) \mid (1, [\,], \varepsilon)$$

3. Inside the else branch the forth element of the list *weight_per_day* is accessed. Therefore the list must contain at least four elements. We can apply the meet of the current assumption domain element associated with the length of the variable *weight_per_day* with the assumption $(Int, [4, \infty])$:

$$len(weight\_per\_day) \rightarrow (Int, [0, \infty]) \sqcap (Int, [4, \infty]) = (Int, [4, \infty])$$

4. Before we exit the else branch we can assume the inverted condition of line 16, that is $day \neq 1$. For this we need to apply the filter operation for each element that is in the map. Using the filter operator of the type domain we can modify the assumption of *day* because we know from the type inference algorithm that *day* must be of type integer. For the range we cannot modify the current assumption because the analysis would need to support disjunctions of intervals to represent all values except 1, which it currently does not. The final assumption for *day* is

$$day \rightarrow (Int, [-\infty, \infty])$$

Because the condition $day \neq 1$ cannot be put into the form $\pm x \pm y \leq c$ we will not add a relation to the relations set.

5. When we exit the else branch we pop the top element of the input assumption stack:

$$inputs : pop((1, [\,], \varepsilon) \mid (1, [\,], \varepsilon)) = (1, [\,], \varepsilon)$$

6. Analysing the if branch of the program is similar to analysing the else branch that started in step 2. For line 17 of the program we can infer the assumption $(Int, [3, \infty])$ for the variable $len(weight\_per\_day)$. The current assumption for the length identifier of *weight_per_day* on this program path is $(Int, [0, \infty])$. When we meet these assumptions we get:

$$len(weight\_per\_day) \rightarrow (Int, [0, \infty]) \sqcap (Int, [3, \infty]) = (Int, [3, \infty])$$

7. Before exiting the if branch we can assume the condition $day == 1$. For this condition we can create the interval element $[1, 1]$. Therefore we can update our map so that

$$day \rightarrow (Int, [1, 1])$$

8. After analysing both branches we have two different program assumption domain elements that need to be merged. To join these elements we apply the join point-wise to the elements in the program assumption domain. The result for the mapping of values to assumptions is as follows:

$$\begin{aligned} weight\_per\_day &\rightarrow (Any, \top) \sqcup (Any, \top) = (Any, \top) \\ len(weight\_per\_day) &\rightarrow (Int, [3, \infty]) \sqcup (Int, [4, \infty]) = (Int, [3, \infty]) \\ day\_string &\rightarrow (Any, \top) \sqcup (Any, \top) = (Any, \top) \\ day &\rightarrow (Int, [1, 1]) \sqcup (Int, [-\infty, \infty]) = (Int, [-\infty, \infty]) \\ other\_day &\rightarrow (Any, \top) \sqcup (Any, \top) = (Any, \top) \\ food &\rightarrow (Any, \top) \sqcup (Any, \top) = (Any, \top) \end{aligned}$$

   The join of the input assumption stack elements works as follows:

   $$inputs : (1, [\,], \varepsilon) \sqcup (1, [\,], \varepsilon) = (1, [\,], \varepsilon)$$

   Finally, the elements of the relations domain are joined as follows:

   $$relations : \{\,\} \sqcup \{\,\} = \{\,\}$$

9. To analyse line 14 we first need to enter the loop body and the else branch so we need to push two new elements on the input stack:

   $$inputs : (1, [\,], \varepsilon) \mid (1, [\,], \varepsilon) \mid (1, [\,], \varepsilon)$$

   From the cast in line 14 we can conclude that the input must be of type float. We can use this information to update the assumption we have for variable $food$ to get the assumption $(Float, [-\infty, \infty])$. To apply the assign operator for the program assumption domain element we first substitute all occurrences of $food$ with the input identifier $id14$. This set is currently empty so nothing has to be done. We then create a new input assumption element $((Float, [-\infty, \infty]), \{\}, id14)$ and add this assumption to the list of assumptions on top layer of the assumption stack:

   $$inputs : (1, [\,], \varepsilon) \mid (1, [\,], \varepsilon) \mid (1, [((Float, [-\infty, \infty]), \{\,\}, id14)], \varepsilon)$$

10. When we exit the branch in line 13 we pop the top element from the stack and get:

    $$inputs : (1, [\,], \varepsilon) \mid (1, [((Float, [-\infty, \infty]), \{\,\}, id14)], \varepsilon)$$

    and assume the condition for this branch:

    $$other\_day \rightarrow (Float, [-\infty, 2])$$

11. The analysis for line 12 works similar and we get:

    $$inputs : (1, [\,], \varepsilon) \mid (1, [((Int, [-\infty, \infty]), \{\,\}, id12)], \varepsilon)$$

    and:

    $$other\_day \rightarrow (Float, [3, \infty])$$

12. When we join these branches we get:

$$inputs : (1, [\,], \varepsilon) \mid (1, [((Float, [-\infty, \infty]), \{\,\}, id12)], \varepsilon)$$

and:

$$other\_day \rightarrow (Float, [-\infty, \infty])$$

13. In line 10 is a raise statement that we do not want to reach. We will use the element $\perp$ as the current program assumption domain element.

14. For the else branch to the if condition on line 9 we can assume the statement $other\_day \geq day$. The filter operation for the program assumption domain will add this relational constraint to the list of relations for this path:

$$relations : \{day - other\_day \leq 0\}$$

15. Because we have the bottom element for the program assumption domain element of the if branch we will just keep the element from the else branch. Therefore we keep the relation $\{day - other\_day \leq 0\}$.

16. Similarly for the if statement in line 7 we will end up with the program assumption domain element of the else branch. This means that we can assume the condition $other\_day \geq 0$ so that we can update the assumption in the mapping:

$$other\_day \rightarrow (Any, \top) \sqcap (Float, [0, \infty]) = (Float, [0, \infty])$$

17. In line 6 we first have to check the input assumption stack for appearances of the variable $other\_day$ that would now be substituted for the input identifier $id6$. This is not the case so we check the relation domain element. There we have the relation $day - other\_day \leq 0$. This relation will be substituted for $day - id6 \leq 0$. Finally we create the input assumption element $((Float, [0, \infty]), \{day - id6 \leq 0\}, id6)$ and add this input assumption to the front of the assumptions list on top element of the stack:

$$\begin{aligned} inputs : (1, [\,], \varepsilon) \mid (1, [((Float, [0, \infty]), \{day - id6 \leq 0\}, id6), \\ ((Float, [-\infty, \infty]), \{\,\}, id12)], \varepsilon) \end{aligned}$$

18. In line 5 we reach the loop header. We will pop the top element of the input assumption stack and create a new multi input assumption element that incorporates all the assumption of the current top layer:

$$\begin{aligned} inputs : (1, [(10, [((Float, [0, \infty]), \{day - id6 \leq 0\}, id6), \\ ((Float, [-\infty, \infty]), \{\,\}, id12)], \varepsilon)], \varepsilon) \end{aligned}$$

19. In line 4 we add a relation:

$$relations : \{day - len(weight\_per\_day) \leq -1\}$$

20. In line 3 we need to substitute all occurrences of *day* with *day_string* and move the assumption of *day* to *day_string*:

$$inputs : (1, [(10,[((Float, [0, \infty]), \{day\_string - id6 \leq 0\}, id6),$$
$$((Float, [-\infty, \infty]), \{\ \}, id12)], \varepsilon)], \varepsilon)$$

$$relations : \{day\_string - len(weight\_per\_day) \leq -1\}$$

$$day\_string \rightarrow (Int, [-\infty, \infty])$$
$$day \rightarrow (Any, \top)$$

21. In line 2 a new input assumption element $((Int, [-\infty, \infty]), \{id2 - len(weight\_per\_day) \leq -1\}, id2)$ is added and the occurrences of *day_string* are substituted:

$$inputs : (1, [((Int, [-\infty, \infty]), \{id2 - len(weight\_per\_day) \leq -1\}, id2),$$
$$(10, [((Float, [0, \infty]), \{id2 - id6 \leq 0\}, id6),$$
$$((Float, [-\infty, \infty]), \{\ \}, id12)], \varepsilon)], \varepsilon)$$

and we need to reset the assumption for *day_string* and the relations element:

$$day\_string \rightarrow (Any, \top)$$

$$relations : \{\ \}$$

22. Finally in line 1 we substitute all occurrences of *weight_per_day* with *id*1 and add the assumption $(3, [((Any, \top), \{\ \}, id1)], ";")$:

$$inputs : (1, [(3, [((Any, \top), \{\ \}, id1)], ";"),$$
$$((Int, [-\infty, \infty]), \{id2 - len(id1) \leq -1\}, id2)$$
$$(10, [((Float, [0, \infty]), \{id2 - id6 \leq 0\}, id6),$$
$$((Float, [-\infty, \infty]), \{\ \}, id12)], \varepsilon)], \varepsilon)$$

This concludes the analysis and the above assumption list is the final precondition that has to be fulfilled by the input data because otherwise there would be an error. It states the following:

The first input is expected to contain at least three values separated by a semicolon. The next input must be of type integer and it must be smaller than the number of elements in the first input. Then there follow two assumption that have to hold ten times consecutively. The first one of those assumptions states that we expect a float that must be greater or equal to zero and the input must be greater or equal to the second input. The second assumption expects a value of type float.

3 Input Checker

The goal of the input checker is to find values in an input file that violate implicit assumptions that are made by a program about the input data and to report the problematic input values to the user. The assumptions that are used to check the input data originate from the static analysis that was described in the previous chapter. The input checker will iterate through the assumptions and determine whether the corresponding input values violate these assumptions.

Input values that violate an assumption should be reported to the user in a way that helps them to understand why the problem happens and how they can fix it so that the program executes successfully. To achieve this we created a graphical user interface that enables the user to correct these input values. The user is presented with an error message that indicates what the problem is and what is expected by the program. The input value can be modified by the user so that it fulfils the assumption made about it.

The next section shows how the communication between the static analysis and the input checker works. Section 3.2 explains how the input checker finds problematic values in an input data file. Section 3.3 presents the graphical user interface and usage of the input checker tool.

3.1 Separation of Static Analysis and Input Checker

To find and report unexpected input values to the user the input checker needs the assumptions that are made by the program about its input data. To get these assumptions the static analysis will run first and hand over the result to the input checker. If the user restarts the checker because an input file has changed or the input checker is run for multiple input data files that belong to the same program, the static analysis and input checker will run again for the same program. In the case where the program is unchanged the static analysis would produce the same assumptions as before.

Running the analysis takes time and executing it again even though the result will be the same is redundant. Therefore it is reasonable to separate the static analysis from the input checker so that they can be run independently. The separation is achieved by storing the assumptions that were found by the static analysis in a JSON file. The input checker can read the JSON file to obtain the assumptions without having to rerun the analysis. This also enables the possibility to manually add further assumptions that will be used by the input checker.

Format of the JSON File

```
[
    [
        {
            "iterations": {
                "expr_var_pos": true,
                "expr_var": ".VAR0",
                "expr_const": 3
            },
            "assmps": [
                {
                    "type_assmp": "Any",
                    "range_assmp": [
                        -Infinity,
                        Infinity
                    ],
                    "relations": [],
                    "id": ".ID=1"
                }
            ],
            "delimiter": ";"
        },
        {
            "type_assmp": "Int",
            "range_assmp": [
                -Infinity,
                Infinity
            ],
            "relations": [
                {
                    "rel_this_pos": true,
                    "rel_this_id": ".ID=2",
                    "rel_other_pos": false,
                    "rel_other_id": "len(.ID=1)",
                    "rel_constant": 1
                }
            ],
            "id": ".ID=2"
        },
        {
            "iterations": {
                "expr_var_pos": true,
                "expr_var": ".VAR0",
                "expr_const": 10
            },
            "assmps": [
                {
                    "type_assmp": "Float",
                    "range_assmp": [
                        0,
                        Infinity
                    ],
                    "relations": [
```

```
53                          {
54                              "rel_this_pos": false,
55                              "rel_this_id": ".ID=6",
56                              "rel_other_pos": true,
57                              "rel_other_id": ".ID=2",
58                              "rel_constant": 0
59                          }
60                      ],
61                      "id": ".ID=6"
62                  },
63                  {
64                      "type_assmp": "Float",
65                      "range_assmp": [
66                          -Infinity,
67                          Infinity
68                      ],
69                      "relations": [],
70                      "id": ".ID=12"
71                  }
72              ],
73              "delimiter": null
74          }
75      ],
76      {
77          "inputs": [
78              ".ID=6",
79              ".VAR0",
80              "len(.ID=1)",
81              ".ID=2"
82          ]
83      }
84  ]
```

Listing 6: Example for the JSON file showing information from the static analysis

Listing 6 shows the JSON file that was produced by the static analyser for the program in Listing 1. It contains all information that is needed by the input checker to be capable of detecting unexpected input values. Lines 2 - 75 contain a list of assumptions that describe the expectations for the input values. In lines 76 - 83 a list of input identifiers is included that shows all identifiers that are used in relational assumptions or in loop iteration expressions. This will be explained in more detail in Section 3.2.

A grammar for the structure of the JSON file is given below:

$$
\begin{aligned}
ID &: \{\text{".}id\text{=}INT\text{"}, \text{".}var0\text{"}\} \\
TYPE &: \{\text{"}int\text{"}, \text{"}float\text{"}, \text{"}any\text{"}\} \\
EXPR &: \{expr\_var\_pos : BOOLEAN, \\
&\quad expr\_var : ID, \\
&\quad expr\_const : INT\} \\
RELATION &: \{rel\_this\_pos : BOOLEAN, \\
&\quad rel\_this\_id : ID, \\
&\quad rel\_other\_pos : BOOLEAN, \\
&\quad rel\_other\_id : ID, \\
&\quad rel\_constant : INT\} \\
ASSMP &: \{type\_assmp : TYPE, \\
&\quad range\_assmp : [INT, INT], \\
&\quad relations : [RELATION^*], \\
&\quad id : ID\} \\
MULTIASSMP &: \{iterations : EXPR, \\
&\quad assmps : [ASSMP^*], \\
&\quad delimiter : STRING\} \\
JSONFILE &: [[(ASSMP|MULTIASSMP)^*], [ID^*]]
\end{aligned}
$$

We explain the elements with the help of the example in Listing 6:

ID Each assumption is given an identifier. It is of the form *.id=INT* where INT is substituted with the line number in which an input was read for which the assumption has to hold. The identifier is used by expressions in $EXPR$ as well. A special identifier *.var0* contains the value zero and is used when an expression is a constant and does not refer to an other input identifier. An example for an id is found in line 17 which refers to the input that is read in line one of the program.

TYPE The types that can be used by the input checker are *int*, $float$ and *any* which refer to the types used by the static analyser. A type *int* is for example used in line 23.

EXPR An expression in $EXPR$ is of the format $\pm x + c$ where x is an ID and c is a constant integer. The elements in $EXPR$ are the following:

expr_var_pos This element decides whether the variable x is positive or negative. If set to *true* the expression is of the form $x + c$, if set to $false$ it will be of the form $-x + c$. In line 5 the variable is positive.

expr_var This element contains the identifier to which this expression refers to. In line 6 the special identifier *.var0* is used to indicate that this expression is a constant expression.

expr_const To describe the constant for the expression of the form $\pm x + c$ the element *expr_const* is used and contains an integer value as for example the value 3 in line 7.

An example for an expression is found starting from line 5. It refers to an expression of the form

$$\pm x + c = +.var0 + 3 = 0 + 3 = 3$$

RELATION A relation is of the form $\pm x \pm y \leq c$ where x and y are identifiers and c is a constant. The elements of $RELATION$ are the following:

rel_this_pos This element decides whether the variable x is positive or negative.

rel_this_id This element contains the identifier to which x refers to.

rel_other_pos This element decides whether the variable y is positive or negative.

rel_other_id This element contains the identifier to which y refers to.

rel_constant This element contains a constant integer value that is used for c.

An example for a relation is found in lines 53 - 59. It is of the form

$$\pm x \pm y \leq c = -.id6 + .id2 \leq 0$$

ASSMP An $ASSMP$ contains information about the non-relational and relational assumptions that are made about an input. In particular the elements are:

id Each assumption contains a unique identifier.

type_assmp The assumption about the type of the input.

range_assmp The assumption about the range of the input.

relations The relational assumptions that include the current input identifier. A relational assumption is always recorded together with the assumption that is used for the input that is read later in the program.

An assumption is for example found in lines 22 - 38. Line 23 shows that the input must be of type integer. The assumption about the range is $[-\infty, \infty]$ as encoded in line 24. The list of relations is found in line 28 and the identifier of this assumptions in line 37.

MULTIASSMP A multi-assumption contains a list of assumptions, where each assumption is used for multiple input value. This happens when inputs are read inside a loop or because an input consists of multiple values that are separated by a delimiter. We distinguish these two cases as follows:

1. Multi-assumption for inputs that are read inside a loop:

delimiter The delimiter is set to *null* because it not necessary in this case.

assmps The list of assumptions used by this multi-assumption.

iterations The number of iterations is in the format of an $EXPR$. It encodes the number of times the assumptions in the *assmps* list have to hold.

An example for a multi-assumption is found in lines 39 - 74. We know that this multi-assumption refers to case 1 because the delimiter is set to *null* in line 73. The list of assumptions is found in line 45 and the number of iterations in line 40.

2. Multi-assumption for an input that contains multiple values separated by a delimiter:

delimiter The delimiter is set to a string that is used to separate the values of the input.

assmps The list of assumptions contains only one assumption that is used for each of the values in the input.

iterations The number specified in *iterations* is in the format of an $EXPR$ that uses the special variable $.var0$ because it has to be a constant number. It encodes the minimum number of elements that we expect when we split the input by the specified delimiter.

An example for this case can be found in lines 3 - 21. The delimiter is set to ";" in line 20, the list of assmps is specified in line 9 and the element that defines the minimum number of expected elements is found in line 4.

JSONFILE The whole JSON file consists of a list of assumptions, respectively multi-assumptions, and a list of input identifiers. The list of assumptions corresponds to the order in which the inputs are read. The list of assumptions is in lines 2 - 75, the list of input identifier in lines 76 - 83.

3.2 Input Checking Algorithm

The input checking algorithm takes as input the list of assumptions $[a_1, ..., a_n]$, that is read from the JSON file. An element a_i is either an assumption or multi-assumption as explained in the previous section. The input checking algorithm will create a list of errors $[e_1, ..., e_m]$ that contains information about unexpected input values and why they cause problems.

Assumptions and Multi-Assumptions We represent an assumption as a tuple

$$(id, type\_assmp, range\_assmp, relations)$$

where the elements correspond to the information read from the JSON file as explained in the previous section. The element *id* is called the *assumption identifier* and is unique for every assumption. It is of the form $.id<INT>$ where $<INT>$ is replaced with the line number of the program in which the input was read for which the assumption has to hold.

A multi-assumption that contains assumptions about inputs that are read in a loop is represented as

$$iterations \times assmps$$

and a multi-assumption that refer to an input with multiple values separated by a delimiter as

$$iterations \times assmps\ with\ delimiter\ D$$

where we add the expression *with delimiter D* and the D is substituted with the delimiter found in the JSON file.

In Listing 6 we have a multi-assumption with delimiter in lines 3 - 21. This assumption is represented as

$$(.var0 + 3) \times [(.id1, Any, [-\infty, \infty], [\,]]\ with\ delimiter\ ";"$$

This means that we expect an input that contains at least three values separated by a semicolon. For each of the elements we do not have any precise information about the type or range.

A multi-assumption that involves inputs that are read in a loop is found in lines 39 - 74. It is represented as

$$\begin{aligned}(.var0 + 10) \times [&(.id6, Float, [0, \infty], [-.id6 + .id2 \leq 0]),\\ &(.id12, Float, [-\infty, \infty], [\,])]\end{aligned}$$

The assumption with identifier $.id6$ states that we expect an input of type float that must be greater or equal to zero. We also have a relational assumption where we expect that the input must be greater or equal to the input that corresponds to the assumption with identifier $.id2$. After this input we expect another input of type float for which we have no precise information about the range and no relational assumptions. These assumptions have to be repeated ten times. This means that if we have an input file with inputs $[l_1, ..., l_{20}]$ we know that the assumption with identifier $.id6$ has to hold for every second input value $[l_1, l_3, l_5, ..., l_{19}]$ and the assumption $.id12$ must hold for input values $[l_2, l_4, l_6, ..., l_{20}]$.

Relational Assumptions The *relations* element of an assumption is a list of relational assumptions that have to hold. To check if a relational assumption holds, the input checker needs the value of both inputs that are involved in the relation. Each relation is only listed once in the JSON file. For an assumption list of the form

$$\begin{aligned}&[..., (id_i, type\_assmp_i, range\_assmp_i, relations_i), ...,\\ &(id_k, type\_assmp_k, range\_assmp_k, relations_k), ...]\end{aligned}$$

a relational assumption that involves id_i and id_k will be recorded in $relations_k$. When we reach the assumption id_k we need to make sure that the input value that corresponds to the assumption id_i is available. This is what we need the list of input identifiers at the end of the JSON file for. Every time the input checker encounters

an assumption whose identifier is found in this list of input identifiers, the input value that corresponds to that assumptions is saved. It can then be recalled when a relational assumptions needs to be verified.

For example in Listing 6 the first two assumptions in lines 3-21 are represented as

$$[(.var0 + 3) \times [(.id1, Any, [-\infty, \infty], [\,]]\ with\ delimiter\ ";",\\ (.id2, Int, [-\infty, \infty], [.id2 - len(.id1) \leq 1])]$$

The second assumptions contains the relational assumption that the second input value is smaller than the number of elements in the first input when it is split by a semicolon. When the input checking algorithm encounters the first assumption it will store the number of elements of that list because it occurs in the list of input identifier written in lines 76 - 83. When checking the second input we can recall the value and verify if the relational assumptions is fulfilled.

Algorithm The whole input checking algorithm is explained below:

1. The input checker reads the content of the JSON file that was created by the static analysis and iterates through the resulting assumption list $1 \times [a_1, ..., a_n]$. An input value dictionary is initialized that will store all values that are needed for relational assumptions or iteration expressions. This dictionary contains an entry for all the elements of the input list that is recorded at the end of the JSON file.
2. If the current assumption list is of the form $N \times [a_1, ..., a_n]$ *with delimiter* D:
   (a) Read the next line from the input data file.
   (b) If the input file has reached the end, the checker creates an error for a missing value and stops checking for this line.
   (c) Evaluate the iteration number N. It is either a constant or dependent on an other input. If it depends on an other input we can recall the value from the input value dictionary. If the number cannot be found there was an error when that input value was checked. The input checker is not yet able to evaluate the number N and will not continue checking further assumptions for this input.
   (d) If the input checker was able to evaluate N: Split the input line by the delimiter D and check if there are at least N values. If not, an error message for the expected number of values is created. The number of elements that were found will be stored in the input value dictionary if it contains an entry for the length of the list with input identifier of the current assumption.
   (e) If no error was created, check all relational assumptions. The input checker expects relational assumptions of a list to be a relation between its length and another value, since we currently only support assumptions that have to hold for the whole list and not assumptions about particular elements of the list. If a relational constraint is violated an error messages will be created.
3. If the current assumption list is of the form $N \times [a_1, ..., a_n]$:

(a) Evaluate the iteration number N. It is either a constant or dependent on an other input. If it depends on an other input we can recall the value from the input value dictionary. If the number cannot be found there was an error when that input value was checked. The input checker is not yet able to evaluate the number N and the input checking algorithm will stop. The input checking cannot continue because we do not know for how many values the current assumptions have to hold. Therefore we cannot identify the next input value that corresponds to the next assumption.

(b) If N was evaluated: Loop through the list of assumptions N times. For each assumption a_i:

   i. If a_i is again of the form $M \times [b_1, ..., b_m]$ go to step 2.

   ii. If a_1 is a single assumption $(id, type\_assmp, range\_assmp, relations)$ with a type and range assumption and a list of relational assumptions:

      A. Read the next line from the input data file.

      B. If the end of the input file is reached the checker creates an error for a missing value. Go to step 3b to check the next assumptions.

      C. If the type is wrong, an error explaining the wrong type is created. Go to step 3b to check the next assumptions.

      D. If the type is correct the value will be added to the input value dictionary if it contains an entry for the input identifier of the current assumption.

      E. If the range is wrong, an error explaining the wrong range is created. Go to step 3b to check the next assumptions.

      F. If there are relational constraints each relation is checked using the input value dictionary that stores the values of the other inputs used in the relation. If a relational assumption is violated a relational error is created.

3.3 Design of the Input Checker Tool

User Interface The problems that are found by the input checker algorithm are presented to the user. The way the errors are presented should help the user to understand what went wrong and how to resolve the issue. To achieve this we decided to create a stand-alone tool with a graphical user interface. A screenshot of the tool is shown in Figure 6 where an error is presented because an input value is of the wrong type. The user is presented one error at a time and the next problem will appear as soon as the current one was fixed. At the top, an error message describing the problem with the input value is displayed. Next to it the total number of discovered errors is shown. On the left hand side an excerpt of the input data file is visible to give the user some context of the location where the error happened. The input field on the right hand side provides the user with the possibility to correct the value. The entry is confirmed by pressing the return key on the keyboard. Next to the entry field the expectations of the input are shown. At the bottom a button can be pressed to open and alter the input file directly.

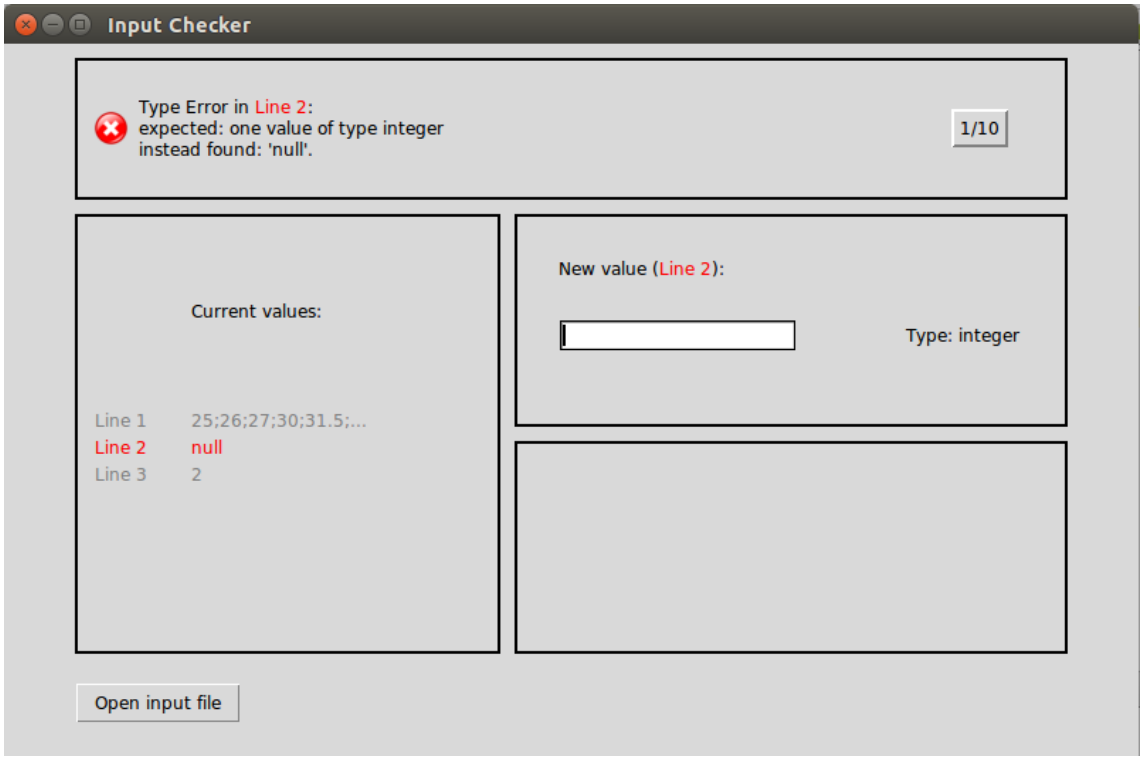


Figure 6: Screenshot of the Input Checker Tool Showing a Type Error

Error Message To help the user understand why an input value caused a problem an error message will be shown in the top part of the user interface. The way of designing these messages is inspired by compiler design. In [10] it is suggested that an error message should be specific, user-directed and complete. More precise, an error should tell the user what was expected but not found, the error message should be written in a context that is comprehensible to the user and the information that is displayed together with the error messages should be adapted depending on the kind of error that occurred. In the case of the input checker this means that we should tell the user the exact location an error was found in the input data file and what we expect of the value. Depending on whether an input is missing, the type of the input is wrong or whether a relation is violated we should adapt the information we display.

We distinguish error messages of the following kind:

Missing An input value is missing because there are less entries in the input file than expected.

Type The type of an input is not as required.

Range The value is outside of an expected range.

Relation A relational assumptions between two values is violated.

For each of them we generate a suitable error message. The general structure of the error messages is:

<Kind of Error> in <location>:

expected: <Violated format>
instead found: <Value found at input location>

where <Kind of Error> is one of the error types mentioned before, <location> is the line number of the input file where the error occurred, <Violated format> tells the user what kind of format we expected from the input value but was violated and <Value found at input location> shows the value that was found at the location.

There is one exception to this format for errors that are created if the end of the input file is reached even though there are still assumptions. This means that the input file is missing some values. To present these errors only the first line of the error message format, namely <Kind of Error> in <location>, is used. For example, if we have the assumptions $[a_1, ..., a_n]$ for the input file $[l_1, ..., l_m]$ and the input checker finds that $n > m$ we will create the following error messages: $\forall i \in [m+1, n] : "Missing\ value\ in\ Line\ i"$.

These error messages are specific, because we tell the user what expectation was not fulfilled by the input value. They are user-directed because we point out the location of the error in the input file and only present information related to the input and not for example the program. For the messages to be complete we show to the user all information we have about the format. This information is not included in the error messages itself but will be displayed next to the entry field where an input value can be modified.

Relational Error The presentation of a relational error is slightly different to the presentation of a non-relation error. The screenshot in Figure 7 shows how a relational error is presented. The context on the left hand side now shows the context of both values that appear in the relation. The user is first asked to provide a value for the first input of the relation, which is always the value of the input location that is further up in the input file. If the relation is not violated any more after confirming the first input value the input checker will display the next error. Otherwise the user is given the opportunity to alter the second value.

Error Ordering As mentioned before we report missing errors for the lines after the end of an input file is reached. However, we do not know where in the input file the inputs are actually missing. Therefore it is crucial that the user first adds the missing values to the file because it can alter the way the assumptions are matched with an input value. To ensure this, problems concerning missing input values are presented to the user first. Afterwards we present errors about the type, those about the range and the last ones listed are the relational errors. This way similar kinds of errors will be presented in a close progression.

Workflow The workflow of the user interface and the input checker is shown in Figure 8. The input checking algorithm is run and if no errors are found the program will be executed using the input file and the output is presented to the user. Otherwise, the first error, sorted by the ordering presented before, is presented to the user. The user can now alter the current input value using the entry field. After pressing the return key the input checker verifies if all assumptions about the current input value are fulfilled by the newly entered value. If this is the case, the input file

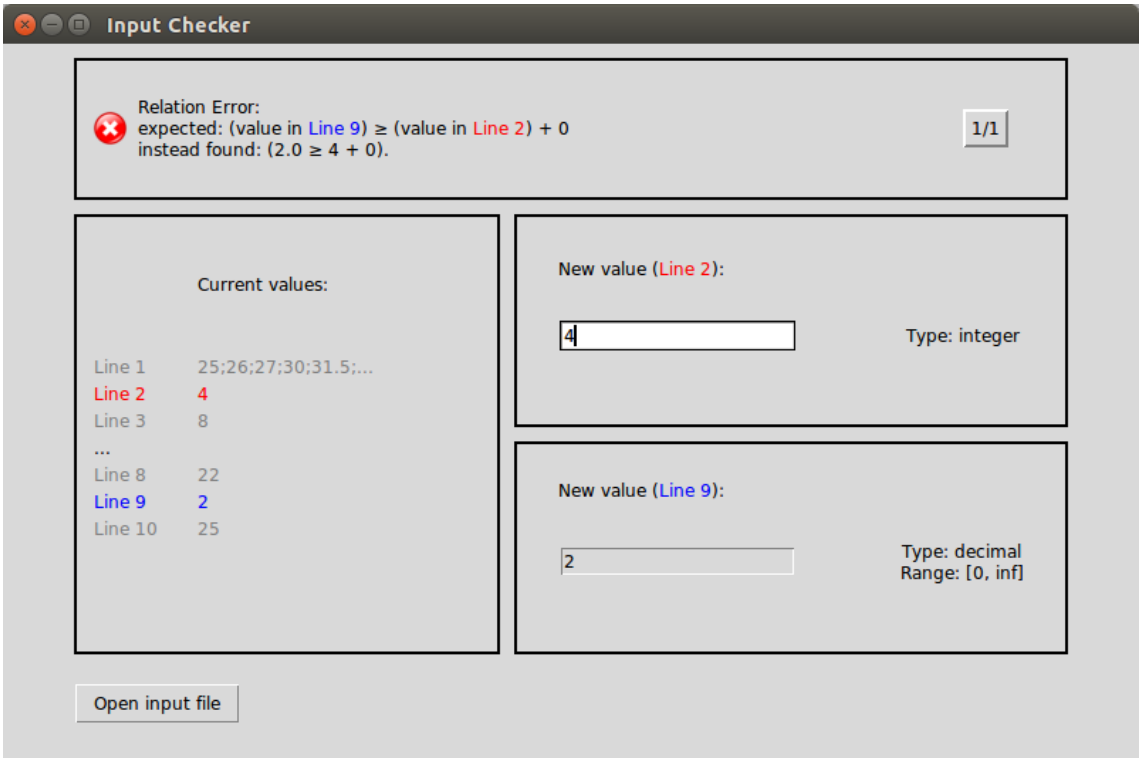


Figure 7: Screenshot of the Input Checker Tool Showing a Relation Error

will be updated with the new value and the input checking algorithm will be rerun. Otherwise, the error message will be updated to present the current problem.

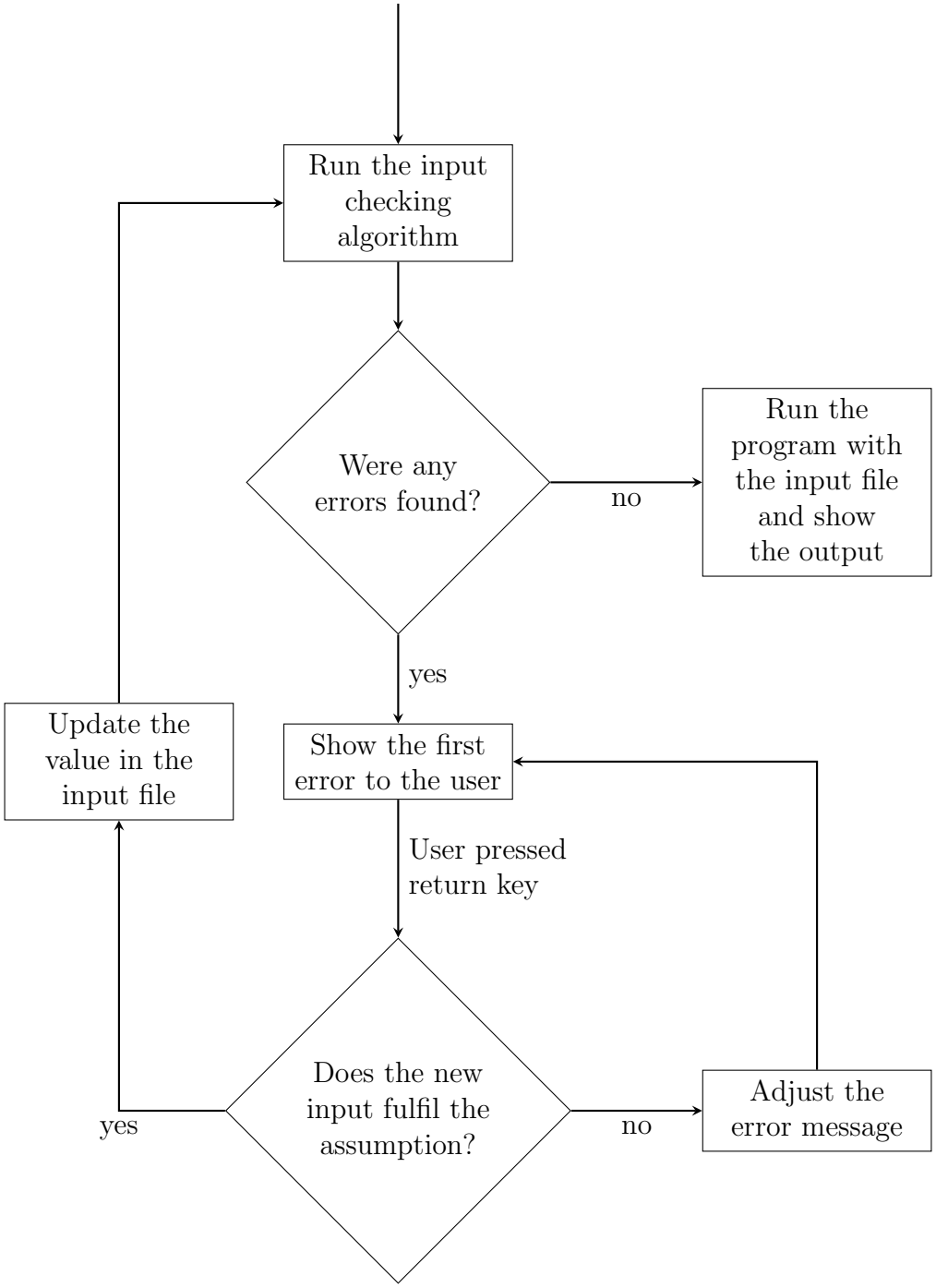


Figure 8: Workflow of the Input Checker.

4 Implementation

The abstract domains defined in Section 2 and the input checker described in Section 3 were implemented as an extension to the Lyra [11] project. The implementation is written in Python.

Program The static analysis takes as input a program written in Python to infer the assumptions the program makes about its input data. The program must include type annotations that can either be added manually or the already existing type inference framework Typpete [12] can be used to generate them. The analysis does not support user defined methods or class structures. Programs that read input values must do so using the call statement *input*().

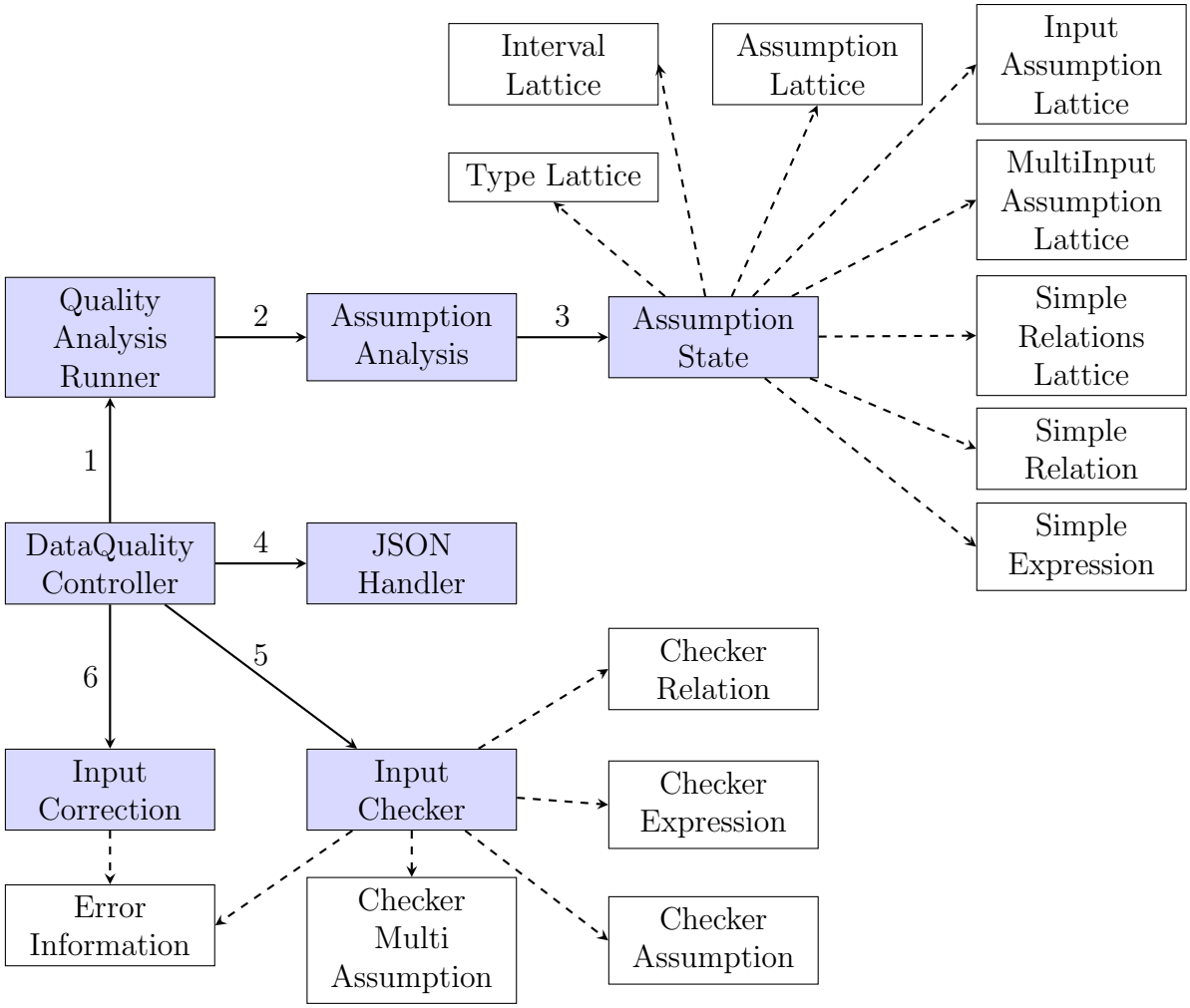


Figure 9: Interaction of the classes implemented for the static analyser and the input checker. Steps 1-4 are only executed if no JSON file for the program exists. Dashed arrows indicate what classes are used by other classes.

Workflow Figure 9 displays the classes that were created for the data quality analysis and shows how they interact with each other. The *DataQualityController* is the

starting point and triggers the different components of the tool. Via the *QualityAnalysisRunner* the static analysis is started. The *AssumptionAnalysis* is a subclass of *Runner* which already existed at the beginning of the project. The *Runner* alongside with other classes build the core framework for the implementation of a static analysis. At the beginning of the analysis a control flow graph representation of the program is created. The backward analysis will traverse the branches of the graph and apply the functions of the abstract domains presented in Section 2.

Each abstract domain presented in this thesis is implemented in a separate class. The interval domain was already part of the Lyra project. Furthermore, the stack mechanism used in the input assumption stack domain already existed so that only the implementations for the *push*() and *pop*() operation had to be added. The program assumption domain is implement in the *AssumptionState* class. The relations domain is implement in the *SimpleRelationsLattice* class. The *SimpleRelation* class stores relations of the form $\pm x \pm y + c \leq 0$ and the *SimpleExpression* class represents expressions of the form $\pm x + c$.

When the static analysis is finished, the resulting assumptions are handed over to the *JSONHandler* in the form of a list of input assumption domain elements, respectively multi input assumption domain elements. They are then written to a JSON file in the format explained in Section 3.1.

The above steps are only executed if no JSON file for the current program exists. Otherwise the checker will not run the analysis and read the assumptions directly from the existing JSON file.

The assumptions from the JSON file are read into an internal representation used by the input checker. The classes that are used for this representation are called *CheckerRelation*, *CheckerExpression*, *CheckerAssumption* and *CheckerMultiAssumption*. The class *CheckerIdentifier* represents an identifier of the form $.id = INT$ where INT is an integer as described in Section 3.1. Subclass *CheckerLengthIdentifier* is used to represent specialised identifiers that refer to the number of elements of an input that contains multiple values separated by a delimiter. Subclass *CheckerZeroIdentifier* represents the variable var_0 that is used when a variable of an expression or relation should be substituted with the value zero.

A list containing objects from the *CheckerAssumption* or *CheckerMultiAssumption* class is given to the input checking algorithm implemented in the *InputChecker* class. The result of this algorithm is a list of $ErrorInformation$ objects that contain the assumptions made about an input, the input value that was found, the location of that value in the file and the error message that is presented to the user. Finally, the *InputCorrection* class contains all implementations needed for the graphical user interface where the result of the input checking algorithm will be presented. The package that is used for the user interface implementation is called *tkinter* [13] and is part of Python.

5 Evaluation

In this section we present the results of the evaluation of our approach. Section 5.1 describes the evaluation that was done for the static analysis, while Section 5.2 presents the user study we conducted to evaluate the input checker.

5.1 Static Analysis

Using program examples, we compared the precondition that results from the static analyser with the assumptions that can be inferred manually. We examined what problematic input values will not be caught by the precondition that was found by the static analyser and why the analyser is not precise enough in those cases. The assumptions that are inferred manually are all assumptions that are made by a program about the type or range (if the type is a numerical one) of an input value and all relational assumptions between input values that can be derived. The programs that are used for the evaluation are examples that were presented throughout the thesis.

Example 1 The assumptions found by the static analysis for the program in Listing 1 were presented in Section 2.4. The precondition that is found by the static analysis is the following:

$$
\begin{aligned}
inputs : (1, [(3, [((Any, \top), \{\,\}, id1)], ";"),\\
&((Int, [-\infty, \infty]), \{id2 - len(id1) \leq -1\}, id2)\\
&(10, [((Float, [0, \infty]), \{id2 - id6 \leq 0\}, id6),\\
&\quad ((Float, [-\infty, \infty]), \{\,\}, id12)], \varepsilon)], \varepsilon)
\end{aligned}
$$

The first input is expected to contain at least three values separated by a semicolon (1). The next input must be of type integer and it must be smaller than the number of elements in the first input (2). Then there follow two assumption that have to hold ten times consecutively. The first one of those assumptions states that we expect a float that must be greater or equal to zero and the input must be greater or equal to the second input (3). The second assumption expects a value of type float (4).

Assumption (4) states that the input must be of type float. The assumption about the type of this input was inferred when the assumptions from the branches of the if-else statement were joined in line 11.

When the program is inspected manually we can make the following assumption about the type of that input:

$$
\begin{aligned}
&if\ id6 > 2 :\\
&\quad (Int, [-\infty, \infty], \{\,\}, id12)\\
&else :\\
&\quad (Float, [-\infty, \infty], \{\,\}, id14)
\end{aligned}
$$

The assumption made by the analysis is less precise than the assumption we can infer manually. If the input that is read in line 6 is greater than zero, but the input read in line 12 is a floating point number the program will yield a *ValueError*. This problem is not caught by the precondition created by the analysis but would be detected by the assumption that we infered manually.

The problem is that the analysis loses precision in this case because the assumption that is created and needs to hold for both branches cannot be precise enough to capture the different assumptions made in each branch.

Similarly an error will be missed if the condition in line 16 is false and line 19 is executed but there are only three elements in the first input. Because of the over-approximation the static analysis can only infer that the input read in line 1 must have at least three elements.

To make the analysis more precise for these cases, a conditional element could be added to the assumptions gathered by the static analyser. The input assumption $id12$ would then look for example like this:

$$(Int, [-\infty, \infty], \{\,\}, id12) \;\; if \;\; id6 > 2 \;\; else \;\; (Float, [-\infty, \infty], \{\,\}, id12)$$

The first assumption $(Int, [-\infty, \infty], \{\,\}, id12)$ has to hold in the case that the input value corresponding to the assumption with identifier $id6$ is greater than two. Otherwise, assumption $(Float, [-\infty, \infty], \{\,\}, id12)$ must hold.

This change would make the analysis more precise but also more complex. The condition that is introduced to the assumption could contain a reference to a program variable which needs to be substitued when an assignment occurs. We would need to define how two conditional assumptions $a_1 \; if \; cond_1 \; else \; b_1$ and $a_2 \; if \; cond_2 \; else \; b_2$ are joined when they appear in different branches. If a conditional assumption $a_1 \; if \; cond_1 \; else \; b_1$ only holds under certain conditions and we want to add this information, we will get a nested construct of the form $(a_1 \; if \; cond_1 \; else \; b_1) \; if \; cond_3 \; else \; (a_2 \; if \; cond_2 \; else \; b_2)$ which is even more complex. Otherwise we need to decide how much nesting the analysis will allow and what to do if the maximum is reached. Analysing these more complex structures will result in a more precise analysis but the execution time will go up.

Example 2 In Listing 2 the static analyser has problems creating a precise assumption because the number of inputs is different for each possible program execution. If the condition $cond == "True"$ is true, one additional input will be read. Otherwise the program reads three inputs that are expected to be of type float. The static analysis would create the following precondition:

$$[(Any, \top, \{\,\}, id1), (Any, \top, \{\,\}, id3), (Float, [-\infty, \infty], \{\,\}, id6)]$$

A precise assumption however would state the following:

$$
\begin{aligned}
&if\ id1 = "True" :\\
&\quad [(Any, \top, \{\,\}, id1), (Any, \top, \{\,\}, id3), (Int, [-\infty, \infty], \{\,\}, id6)]\\
&else :\\
&\quad [(Any, \top, \{\,\}, id1), (Any, \top, \{\,\}, id3), (3, [(Float, [-\infty, \infty], \{\,\}, id6)]),\\
&\quad (Int, [-\infty, \infty], id7)]
\end{aligned}
$$

As in the example before, the analysis loses precision because the assumptions of the two branches of an if-else statement are merged. In contrast to the previous example not only the assumptions made about each input value are approximated but also the expected amount of input values.

If we have different assumptions list depending on a condition and want to add this information to the analysis we have similar problems as before. But this time the analysis does not only have to handle two possible assumptions that are separated by a condition but two different assumption lists. For each assign statement in the program both assumption lists need to be scanned for program variables that need to be substituted. Each time a new input is read, both assumption lists will get larger. We again need to answer the question of how much nesting we allow. All this additions make the analysis more precise but also more complex and the time the analysis needs to analyse the program will increase.

Example 3 For Listing 8 the following assumptions are inferred by the static analyser:

$$
\begin{aligned}
&[(Int, [-\infty, \infty], \{\,\}, id1),\\
&\ id1 \times [(Float, [-\infty, \infty], \{\,\}, id3),\\
&\qquad (Int, [-\infty, \infty], \{\,\}, id4),\\
&\qquad id4 \times [(Int, [-\infty, \infty], \{-id3 + id7 \leq 0\}, id7),\\
&\qquad\quad (Float, [0, \infty], \{\,\}, id12)]]]
\end{aligned}
$$

In line 16 of the program, there is a division where the variable *speed* is used as a divisor. The value stored in *speed* must be different from zero, otherwise a *ZeroDivisonError* is risen by the program. The assumption the analysis gathered for the input value that is stored in the variable *speed* is the assumption with identifier $id12$. The range assumption the analysis infers is $[0, \infty]$. This assumption was gathered by the analysis in line 13, where the inverted branch condition can be assumed because we do not want to reach the error that would be raised in line 14. The range assumption $[0, \infty]$ is not precise enough because it should not include the value zero. The reason why the analysis does not capture the assumption that the value cannot be zero in line 16, is that would need an interval $[-\infty, \infty] \setminus \{0\}$ which can only be captured as $[-\infty, -1] \cup [1, \infty]$ and not as a single interval.

The assumption domain element could be extended to include disjunctions of intervals. The analysis could then capture the assumption in line 16 that the variable *speed* cannot be zero.

In line 22 there is a another division. This time the variable *max_time* cannot be zero. To be sure that *max_time* is not zero, the loop in line 6 must execute at least once because otherwise *max_time* will be zero because it was initialised with that value. The loop will execute at least once if $n > 0$. The assumption the analysis gathered for the input that is stored in the variable n is $(Int, [-\infty, \infty], \{\}, id4)$. The analysis did not capture the problem since the range assumption states $[-\infty, \infty]$.

Furthermore, the computation in line 16 must be greater than zero at least once so that the variable *max_time* is updated in line 18. The goal is to make sure that line 22 is not reached without reaching line 18 first. To achieve this we can conclude the following:

$$\frac{distance - position}{speed} > 0 \tag{1}$$

$$\Rightarrow \quad distance - position > 0 \tag{2}$$

$$\Rightarrow \quad distance > position \tag{3}$$

The variable *speed* can be removed in equation (2) because line 22 will not be reached if *speed* is smaller or equal to zero. The reason is that there would be an error in line 14 if speed is smaller than zero and and error in line 16 if it is equal to zero. The analysis already captures the relational assumption that $distance \geq position$ because of line 9, but would miss errors where $distance = position$, which means that the input read in line 3 is equal to all subsequent inputs that are read in the loop in line 7.

5.2 Input Checker

To evaluate the usability and usefulness of the input checker and to gain feedback for improvements we conducted a user study. We invited four participants to use the input checker tool and to state their opinion about it. The next section explains the method in more detail and Section 5.2.2 shows the results of the user study.

5.2.1 Method

Each participant has to complete two tasks during the user study. The main goal of both tasks is to successfully run a data analysis program, which means that the program shows the output of the program and not an error message. They are not allowed to alter the program implementation but can change the content of the input files that are used by the programs. In the first task they need to complete the task without the input checker tool. They can run the program, see the output or error given by the program and are permitted to look at the source code of the program. In the second task they work with the input checker tool. After each task the participants are given a questionnaire containing questions using Likert Scales [14] that are used to let participants rate how much they agree with a statement. Open questions are used to ask the participant about their opinions.

The complete task description and questionnaires used during the study can be found in the appendix.

Programs We collected two different programs and input files that are used for the tasks the participants need to complete. The program in Listing 7 was taken from a homework exercise solution [15] and the program in Listing 8 is from a coding competition [16]. The code was rewritten so that it contains only instructions supported by the analysis. Further code was added to create more diverse preconditions. The input file for the first program was created from scratch, for the second scenario we found input data on the coding competition website. We modified both input files to include unexpected data values. To achieve this we created a program that randomly introduced around 20 errors in both files. Those errors include swapping an integer with a letter and converting an integer to a float by adding *.0*. Furthermore, relational errors were introduced. For the program in Listing 7 this was done by altering the number of comma separated elements of the inputs that are read in line 37. Either the last element of the list was added again or the last element was removed. Because of these changes the program would run into an error in line 39 or line 41. In Listing 8 an error will occur in line 9 if $distance < position$. This means that the error will occur if the the input read in line 3 is smaller than the input read in line 7. To introduce a relational error, inputs that are read in line 7 where altered so that their value is greater than the value read in line 3.

```
student_names: List[str] = ["", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", "", "", "", "", "",
                            "", "", "", ""]
hw_avgs: List[float] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                        0, 0, 0, 0]
test_grades: List[int] = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0]

subjects: List[str] = input().split(";")
for i in range(100):
    student_name: str = input()
    hw_grades: List[str] = input().split(",")
    if len(subjects) < len(hw_grades):
        raise InvalidFormatException
    if len(subjects) > len(hw_grades):
        raise InvalidFormatException
    hw_sum: float = 0
    for j in range(len(hw_grades)):
        hw_sum: float = hw_sum + float(hw_grades[j])
    test_grade: int = int(input())
    student_names[i]: str = student_name
    hw_avgs[i]: float = hw_sum / len(hw_grades)
    test_grades[i]: int = (test_grade + hw_avgs[i]) / 2

for i in range(len(student_names)):
    print("Student name:")
    print(student_names[i])
    print("Homework average:")
    print(hw_avgs[i])
    print("Test grade:")
```

```
56      print(test_grades[i])
57      print("")
```

Listing 7: Program for scenario 1

```
1  cases: int = int(input())
2  for cc in range(cases):
3      distance: float = float(input())
4      n: int = int(input())
5      max_time: float = 0
6      for i in range(n):
7          position: int = int(input())
8          if distance < position:
9              raise ValueError("The distance cannot "
10                              "be smaller than the "
11                              "current position.")
12         speed: float = float(input())
13         if speed < 0:
14             raise ValueError("The speed cannot "
15                              "be negative.")
16         val: float = (distance - position) / speed
17         if val > max_time:
18             max_time: int = val
19     print('Trip #')
20     print(cc+1)
21     print("with speed")
22     print(distance / max_time)
23     print("")
```

Listing 8: Program for scenario 2

Scenarios To provide the user with a reason to complete the tasks and to give some context we prepared the following two scenarios:

Scenario 1: Grades You are the principal of Washington High School. To evaluate the performance of this year's students you asked all of the teachers to send you a report containing the grades of all homework exercises and the grade of the final test of each student in their class. There are 100 students and the grades range from 0 to 100.
From your IT department you received a program that uses this information and shows you a summary of every student's name, their average homework score and their final grade that is calculated as $\frac{homework\ average + test\ score}{2}$. You now try to run the program using the data you received from the teachers to get the desired report of the 100 students.

Example Output:

Student name:
Aaliyah
Homework average:
40.833333333333336
Final grade:
68.41666666666667

Student name:
Abigail
Homework average:
65.0
Final grade:
82.0

Student name:
Addison
Homework average:
38.166666666666664
Final grade:
23.083333333333332

Scenario 2: Self Driving Cars You are working as a driving simulation engineer for a company that builds self driving cars. To test their ability to drive accident free in daily traffic you created a test scenario. You came up with 100 schedules, each with a different route and amount of cars on the street. Each car starts at a different location on the route and drives with a designated speed. If one car encounters another one it will slow down and match the speed of the car in front of it.
To ensure a smooth drive from the start to the end of the route you want your personal car to travel at the same speed for the whole trip without having to slow down because of an other car. You ask one of your colleagues to create a program that calculates the speed you have to choose given the distance of the route, the amount of cars on the street and their starting points and speeds.
Today the program was installed on your computer and you just finished to complete the simulation schedule. You now want to run the program using the schedule as input to get the information you want.

Example Output:

Trip #
1
with speed
101.0

Trip #
2
with speed
100.0

Trip #
3
with speed
379.7420306226183

Scenario 1 is used for the program in Listing 7 and scenario 2 for the program in Listing 8.

Task 1: Program Error Messages For the first task the participant is given a written explanation of one of the scenarios and an explanation on how to execute the task. The goal is to successfully run the program so that it does not finish with an error message. To achieve this goal they can only alter the content of the input file. They can run the program as many times as they want to see the output or the error given by the program. The user can also look at the program to get information about what went wrong. As soon as they completed the goal or after a maximum of eight minutes the participant is given a questionnaire. They will rate how difficult the taske was for them. Also, there are open questions about what they would expect from a tool that should help them make the input correction process easier. The task description and questionnaire can be found in the appendix.

Task 2: Input Checker The goal of this task is again to successfully run the program but this time they use the input checker tool. They have eight minutes to complete this task and should answer a questionnaire afterwards. We ask them to rate how usable and useful the tool was in achieving their goal and we want to know if the tool met the expectations they had after the first task and if they can give suggestions on how to improve it. The task description and questionnaire can be found in the appendix.

Participants We chose to conduct the study using two students or graduates with a background in computer science (P_3 and P_4) and two participants with a background not in computer science, namely biology (P_1 and P_2). To avoid a biased outcome because we use different scenarios for the two task we created different combinations of participant's backgrounds and scenarios that are used in the first, respectively second, task. The combination of scenarios $S_{[1,2]}$ participants $P_{[1-4]}$ is as shown in Table 1.

	S_1	S_2
P_1	Python error only	with Tool
P_2	with Tool	Python error only
P_3	Python error only	with Tool
P_4	with Tool	Python error only

Table 1: Combination of participants and scenarios for the user study.

5.2.2 Results

Preliminary Questionnaire At the beginning of the study a preliminary questionnaire was given to the four participants to provide their age, to rate their programming experience on a scale from one to five and they were asked about their profession respectively field of study. The age of the participants ranged from 24 to 28. Two of them are graduates in computer science and classified their programming experience to be on an expert level. One person has a background in

bioinformatics and rated their programming experience with a four out of five. The fourth participant has no experience in computer programming and a background in biochemistry.

Execution Time The time to complete the tasks was limited to eight minutes. Figure 10 shows the time the participants took to complete each task. Most participants managed to complete the task with the input checker in less time than the first task. The participant that did not finish the second task in the given time frame managed to fix 13 out of 17 input value problems. The last four errors were completed in thirty seconds after the eight minutes have passed.

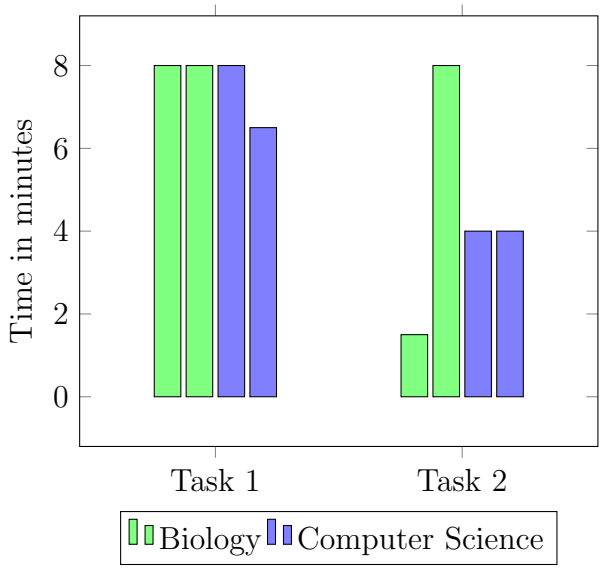


Figure 10: The time that the four participants spent to complete task 1 and task 2.

Results of Likert Scale Questions The Likert Scale questions that were asked in the first task were used similarly for the questionnaire of the second task. We will present how the answers changed from the first to the second task and give an analysis of the results:

Figure 11 shows how much the participants agreed with the statement *"I feel frustrated"*. The colors of the bars represent the participants that have a background in biology, respectively computer science. Most people felt less frustrated when using the input checker tool. The participant that agrees with the statement for the second task, felt neutral in the first task. The reason is that they used to work with the input file directly and it took them some time to familiarise themselves with the input checker tool. For example, it was not clear from the design of the tool that the return key should be pressed to check the newly entered value.

In Figure 12 participants had strong and different opinions whether the first task was easy to solve. The participants that strongly disagreed with the statement were the ones that were given the first scenario, the users that strongly agreed were given the second scenario for the first task. The input file of the first scenario was more structured and had less entries than the second one. Both users rated their

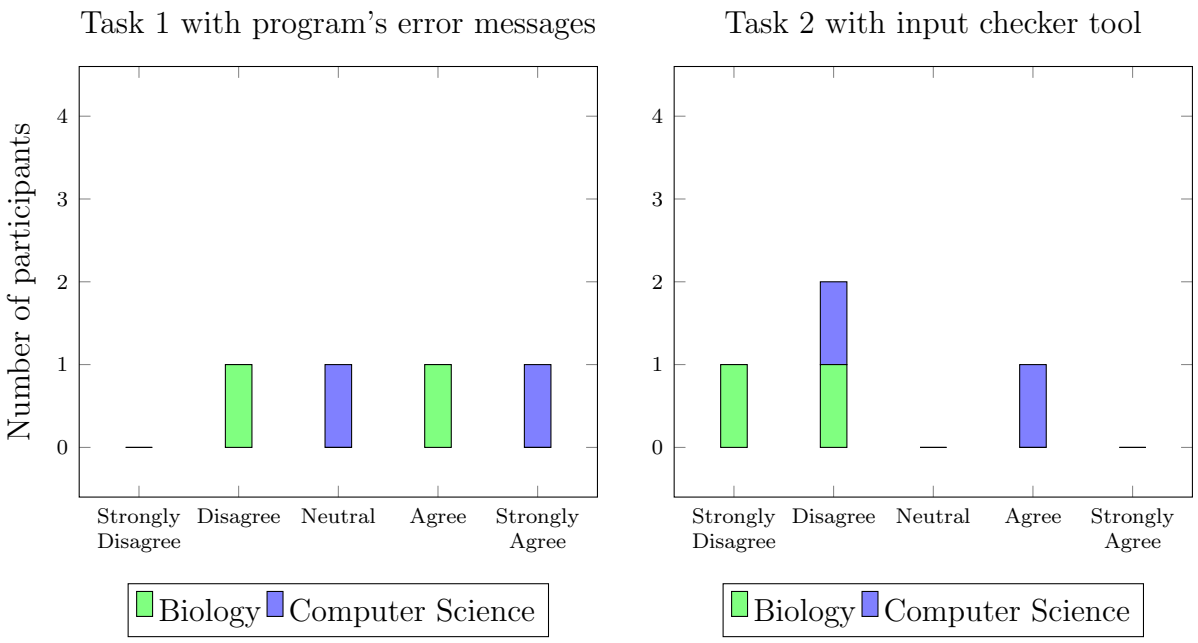


Figure 11: Answers to the statement *"I feel frustrated"*.

programming experience between four and five, so they knew how to interpret an error message, they were able to understand the program and one of them even used a regular expression to scan the input file for errors. For the second task all participants agree that the task was easy to solve.

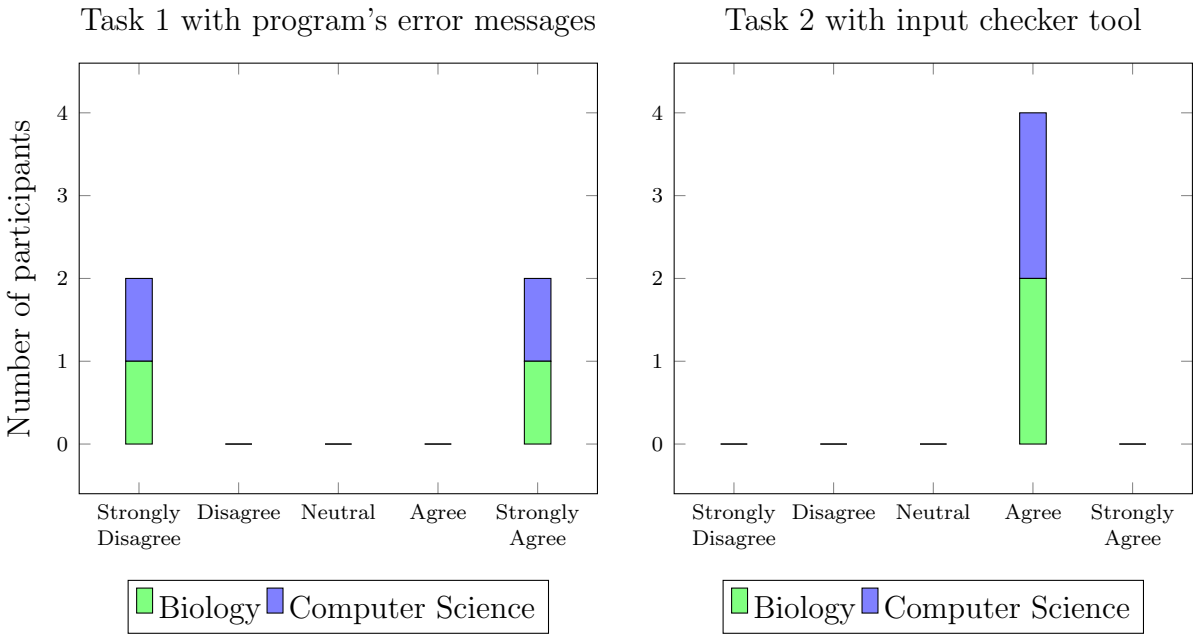


Figure 12: Answers to the statement *"The task was easy to solve"*.

Figure 13 shows the ratings for the statement *"The error messages helped me to understand what was wrong with the input"* for the first task and the ratings for the statement *"The input checker helped me to understand what was wrong with the*

input" for the second task. Most participants agreed that the information given by the input checker made it more obvious what was wrong with the input. The person that disagreed with the statement for the second task, strongly disagreed with the statement in the first task. They were unsure what the correct input should be because they had too little knowledge about the actual content of the data file. They would like to see more suggestions about the content of the input data. However, they also agreed that this is very difficult to do.

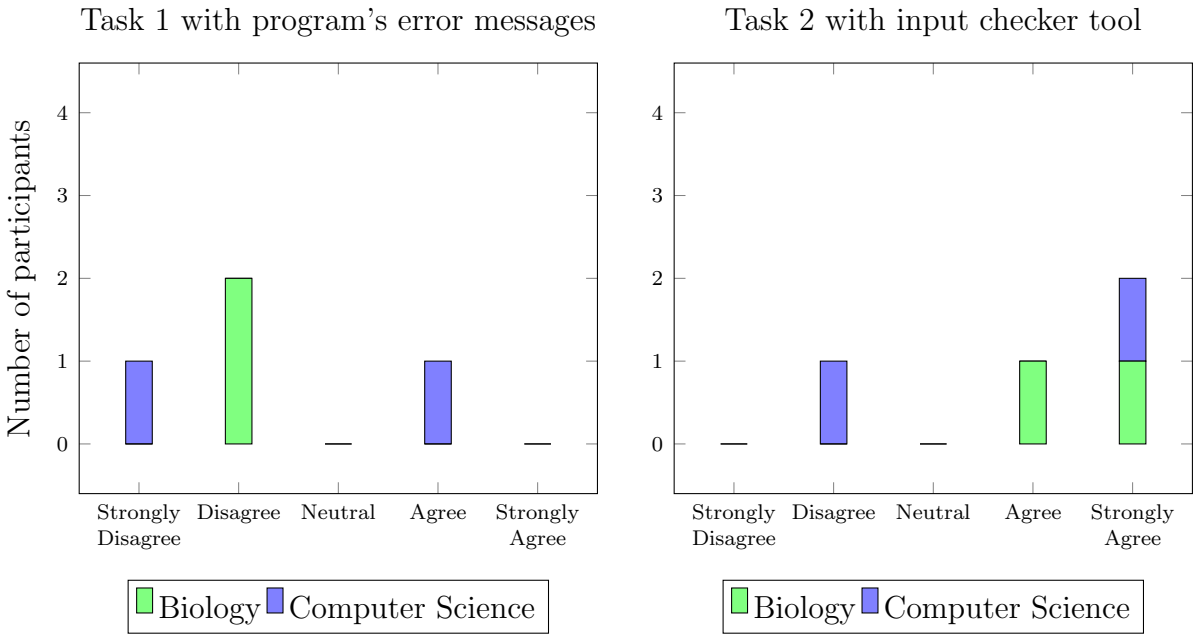


Figure 13: Answers to the statement *"The error messages (respectively the input checker) helped me to understand what was wrong with the input"*.

Figures 14, 16 and 15 show how the participants agreed with statements that were asked specifically about the input checker. Figure 14 shows that most of the participants found the error correction task easier to solve with the input checker than with the error messages given by the program.

Figure 15 shows that most of them would use such a tool in the future.

Most participants agreed that the tool is intuitive to use as seen in Figure 16. One user disagreed because they are more familiar with working on an input file directly.

Difficulty of Task 1 The participants were asked what part of the first task took most of their time and what the difficulties were if they had any. One participant had problems to understand the error message at all and found the task difficult because they *"could not see which line in the input was causing the problem"*. Even if they were able to understand the error message they had to spent time scanning the input manually to find the value that was causing a problem.

Expectations All participants agreed that a tool that helps to find and fix errors in the input file would be useful. They expect a tool that is *"intuitive and easy to use"* and the tool should *"show me where the data is wrong and why"*.

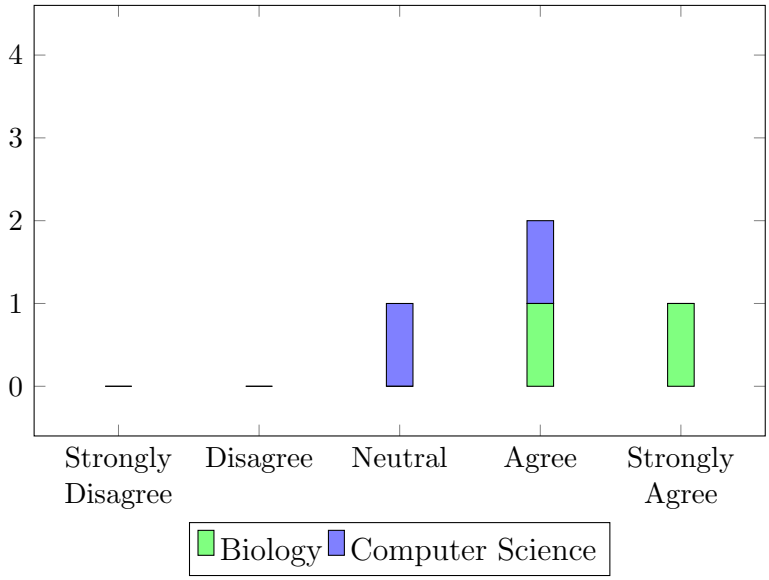


Figure 14: Answers to the statement *"Fixing data with the input checker is easier than doing error correction with the program's error messages only"*.

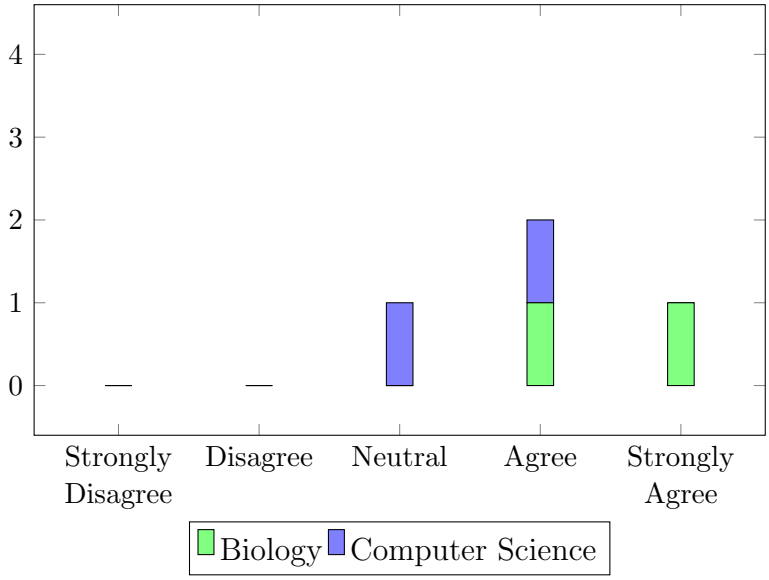


Figure 15: Answers to the statement *"I can imagine using the input checker in the future"*.

Impression of Input Checker Tool The tool was perceived as *"fast and easy to use"* and the participants liked that the tool shows *"the exact source of the error"*. The participant without programming experience indicated that the input checker tool serves as an intermediate between the technical program language and a person that has no background in computer science.

All participants stated that the tool met the expectations they established after the first task. The input checker helped them to *"resolve simple issues fast"* and to *"correct the error and understand how to do so"*. They would however like to see

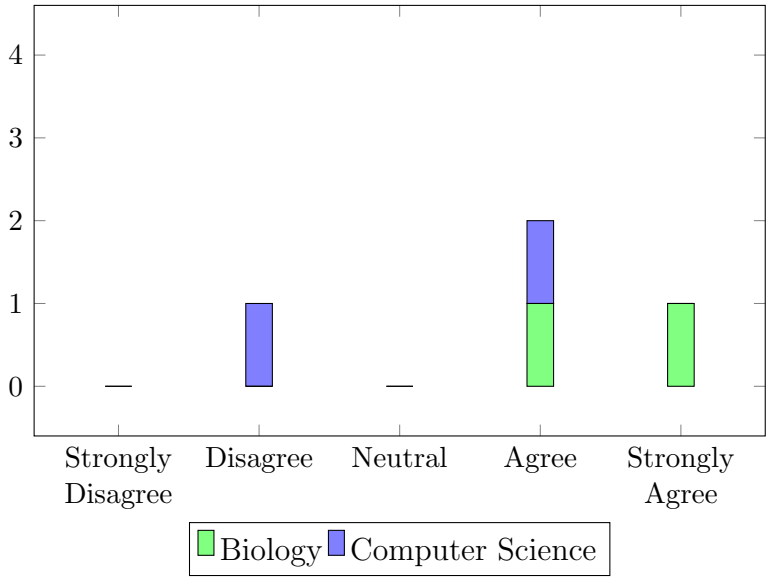


Figure 16: Answers to the statement *"The design of the input checker is intuitive"*.

how the tool deals with more complex structures like expected formats of strings.

Conclusion When solving the first task most participants had an idea how they would tackle the problem. They knew how to decipher the error message and how they could find the problem in the input data. Even though it took them a long time and not everyone managed to complete the problem in the given amount of time they felt familiar with this kind of procedure.

When the participants started the input checker tool they needed some time to get acquainted with it and it was not clear that the entry is submitted by pressing the return key. They were looking for a button to continue. After some initial trial and error the participants got more accustomed to how the tool works and solved the remaining problems fast.

On the one hand it was more intuitive for the participants to work with the input file directly. On the other hand they managed to solve the task faster with the tool and they liked that the tool points out the locations of the problematic input values and what causes the problem. Therefore we argue that the tool would serve much better as a plug-in for a text editor or development environment. The user can work in a familiar environment but would still get the benefits from the input checker tool. This could for example be done by highlighting the problematic input values with colours and showing a hint or popup message that describes the expected format to the user.

6 Conclusion and Future Work

In this thesis we presented the design of a static analysis that automatically infers implicit assumptions that a computer program makes about its input data. The analysis can infer non-relational assumptions about the type of an input and the range if the input is a numerical one. Furthermore, we adapted the initial design to also include relational assumptions.

We created an algorithm that iterates through an input file and checks if the content matches the assumptions that were discovered by the static analysis we developed. If a value violates an assumption the problem is reported to the user. We report missing content, values that do not meet the type or range requirements and violations of relational assumptions.

A tool was developed that includes the implementation of the abstract domains described in this thesis as well as the input checker algorithm. The usefulness of the input checker was evaluated and we demonstrated how the tool supports users to find and fix errors in input data.

To improve the precision of the static analysis and the usability of the input checker, our approach could be extended in the following directions:

Conditional Assumptions To make the analysis more precise, conditional elements could be added to the assumption domains as described in Section 5.1. These elements indicate that an assumption is only valid if a certain condition is met. With this extensions the analysis can keep assumptions that only hold under certain conditions, for example assumptions that are only valid if a loop condition or if-statement evaluates to true.

Look Ahead Input Checking When the input checker detects that input values are missing it cannot make any conclusions about the location of the missing value. A look ahead algorithm could deliver more insight where that location might be. If there is a point in the input file after which many violations of assumptions occur one could conclude that it is because there was a missing value and inputs are matched with the wrong assumptions. A look ahead algorithm would skip some assumptions at that point to see if there are now less violations of the assumptions for the input values that follow.

After an initial check of the input, the input checker can report the number of missing values as the number of assumptions that are left when the end of the input file was reached. If we have assumptions $[a_1, ..., a_n]$ and input file $[l_1, ..., l_m]$ the number of missing values is $n - m$. The input checker could then repeat the checking to find out where those missing values are. Upon encountering a number of subsequent violations of assumptions $[a_i, ..., a_{i+k}]$, the input checker could go back to the point a_{i-1} where the last assumption was fulfilled. It could then skip one assumption and continue matching the next assumption a_{i+1} with the next input value l_i. If the assumptions $[a_{i+1}, ...]$ are now fulfilled that were violated before, we would conclude that there is a missing input value between input values l_{i-1} and l_i. If however the next input value l_i violates assumption a_{i+1}, more assumptions might need to be skipped but not more than the amount of values that is actually missing.

Input Checker Usability In Section 5.2.2 the results of the input checker user study revealed that the design of the graphical user interface has to be improved. As suggested, the input checker could be integrated into a text editor environment to be used as a plugin rather than a stand-alone tool.

References

[1] Joseph M Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

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

[3] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[4] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[5] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[6] Azadeh Farzan, Matthias Heizmann, Jochen Hoenicke, Zachary Kincaid, and Andreas Podelski. Automated program verification. In *International Conference on Language and Automata Theory and Applications*, pages 25–46. Springer, 2015.

[7] Daniel W. Barowy, Dimitar Gochev, and Emery D. Berger. Checkcell: Data debugging for spreadsheets. *SIGPLAN Not.*, 49(10):507–523, October 2014.

[8] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 128–148, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

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

[10] Cornelis HA Koster. Error reporting, error treatment, and error correction in algol translationpart 1. In *GI. Gesellschaft für Informatik eV 2. Jahrestagung*, pages 179–187. Springer, 1973.

[11] Lyra project. `http://www.pm.inf.ethz.ch/research/lyra.html`. Accessed: 16.03.2018.

[12] Typpete project github repository. `https://github.com/caterinaurban/Typpete/tree/artifact`. Accessed: 16.03.2018.

[13] Graphical user interfaces with tk. `https://docs.python.org/3/library/tk.html`. Accessed: 04.03.2018.

[14] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[15] Python programming for engineers, tel-aviv university, spring 2016. http://courses.cs.tau.ac.il/pyProg/1516b/index.html. Accessed: 23.02.2018.

[16] Google code jam. https://code.google.com/codejam/. Accessed: 23.02.2018.

Appendix: Input Checker Study

Scenario 1: Grades

You are the principal of Washington High School. To evaluate the performance of this year's students you asked all of the teachers to send you a report containing the grades of all homework exercises and the grade of the final test of each student in their class. There are 100 students and the grades range from 0 to 100.

From your IT department you received a program that uses this information and shows you a summary of every student's name, their average homework score and their final grade that is calculated as $\frac{homework\ average + test\ score}{2}$. You now try to run the program using the data you received from the teachers to get the desired report of the 100 students. An example output is shown below. Details how to run the program follow on the next page.

Example Output:

Student name:
Aaliyah
Homework average:
40.833333333333336
Final grade:
68.41666666666667

Student name:
Abigail
Homework average:
65.0
Final grade:
82.0

Student name:
Addison
Homework average:
38.166666666666664
Final grade:
23.083333333333332

Scenario 2: Self Driving Cars

You are working as a driving simulation engineer for a company that builds self driving cars. To test their ability to drive accident free in daily traffic you created a test scenario. You came up with 100 schedules, each with a different route and amount of cars on the street. Each car starts at a different location on the route and drives with a designated speed. If one car encounters an other one it will slow down and match the speed of the car in front of it.

To ensure a smooth drive from the start to the end of the route you want your personal car to travel at the same speed for the whole trip without having to slow down because of an other car. You ask one of your colleagues to create a program that calculates the speed you have to choose given the distance of the route, the amount of cars on the street and their starting points and speeds.

Today the program was installed on your computer and you just finished to complete the simulation schedule. You now want to run the program using the schedule as input to get the information you want. An example output is shown below. Please follow the instructions on the next page.

Example Output:

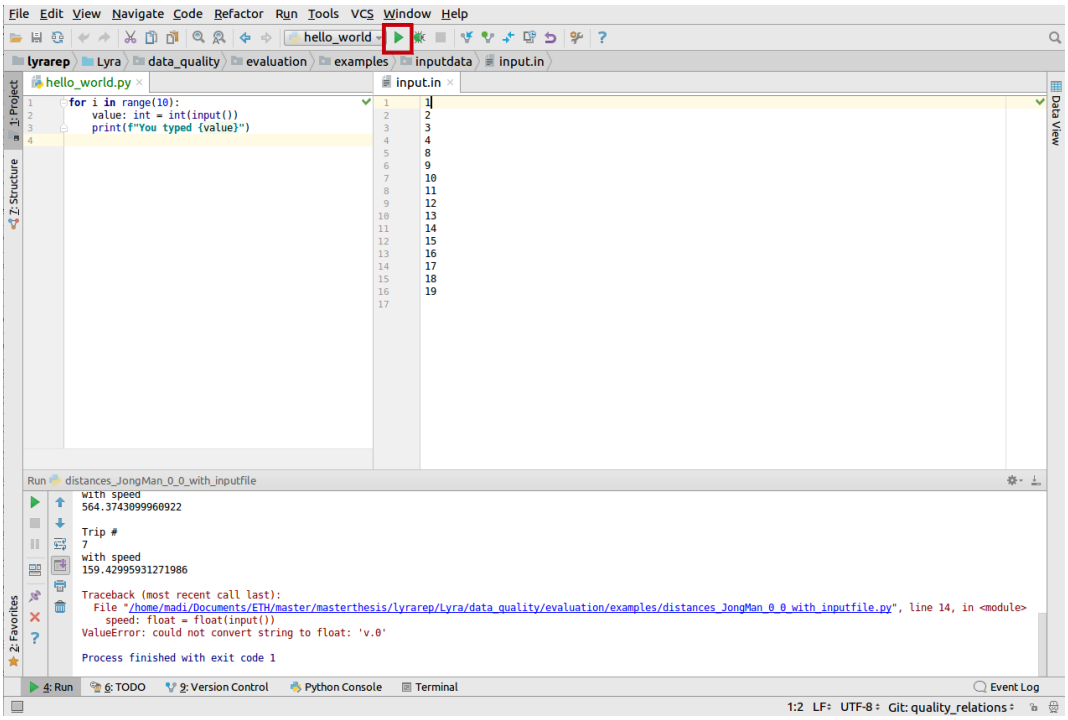
```
Trip #
1
with speed
101.0

Trip #
2
with speed
100.0

Trip #
3
with speed
379.7420306226183
```

Task Explanation for Working with Python's Error Messages Only

You can run the program using the green *Run* button as indicated (with a red frame) in the picture below or by pressing Ctrl+F5. In the top half of the screen you can see the program on the left hand side, your input data on the right side. In the bottom half you see the output of the program after you run it. The output is written with a black font, an error will be shown in a red font.

The goal of this task is to get the desired output from the program as described in the scenario before. You can assume that the program works correctly and if there are any errors then there is a problem with the input data. You are only allowed to alter the input data file. You must **not** alter the program.

You are not allowed to ask questions concerning the error messages or the input data file. You have 8 minutes to complete this task. Please tell me when you are ready.



Task Explanation for Working with the Input Checker

You can again run the program using the green *Run* button or by pressing Ctrl+F5. When you start the program the input checker tool will open up and guide you through the error correction process.

The goal of this task is to get the desired output from the program as described in the scenario before. You can assume that the program works correctly and if there are any errors then there is a problem with the input data. You are only allowed to alter the input data values using the input checker tool. This time you are not allowed to look at the program.

You are not allowed to ask any questions concerning the tool. You have 8 minutes to complete this task. Please tell me when you are ready.

Questionnaire

Please answer the following questions:

What is your age?

_____

What is your profession or field of study?

_____

How would you rate your programming experience?

	1	2	3	4	5	
No experience	☐	☐	☐	☐	☐	Expert

Questionnaire for Task with the Program's Error Messages

1. Please rate how much you agree with the following statements:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I feel frustrated	☐	☐	☐	☐	☐
The task was easy to solve	☐	☐	☐	☐	☐
The error messages helped me to understand what was wrong with the input	☐	☐	☐	☐	☐

2. Please answer the following questions:

   (a) What part of this task took most of your time? If you had any trouble completing the task: What made it difficult?

Please turn the page over.

(b) Do you think it would be useful to have a tool that helps you to find and fix problems in the input data? If yes, what are your expectations of such a tool? What would make it easier to solve the task of input data correction?

Questionnaire for Task with Input Checker Tool

1. Please rate how much you agree with the following statements:

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I feel frustrated	☐	☐	☐	☐	☐
The task was easy to solve	☐	☐	☐	☐	☐
The input checker helped me to understand what was wrong with the input	☐	☐	☐	☐	☐
Fixing data with the input checker is easier than doing error correction with the program's error messages only	☐	☐	☐	☐	☐
The design of the input checker is intuitive	☐	☐	☐	☐	☐
I can imagine using the input checker in the future	☐	☐	☐	☐	☐

2. Please answer the following questions:

   (a) What did you like about the input checker?

Please turn the page over.

(b) What could be improved to make the input checker more helpful?

(c) If you had some expectations about the tool, what expectations were fulfilled? What were not?

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Automated Generation of Data Quality Checks

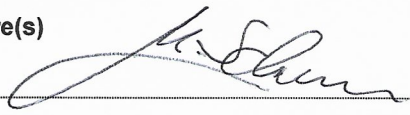
Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schumacher	Madelin

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 17.03.2018	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.