Master Thesis

Automated tutoring of massive open online courses

Author(s):
Zeller, Christine

Publication Date:
2014

Permanent Link:
https://doi.org/10.3929/ethz-a-010255509

Rights / License:
In Copyright - Non-Commercial Use Permitted
Automated Tutoring of Massive Open Online Courses

Christine Zeller

Master Thesis
September 2014

Supervised by
Veselin Raychev
Prof. Dr. Martin Vechev

ETH Zürich

Software Reliability Lab
Abstract

For massive open online courses, the amount of programs submitted tends to be too big compared to the number of people tutoring the course. In order to provide feedback, the organizers rely on automatic assessment through a test suite, and report the output. Novice programmers however struggle to understand error messages, and require more detailed instructions. We therefore present in this thesis a new approach at automatically fixing incorrect programs and providing the programmers with individual feedback, without the need for test sets and specification.

We combine abstract interpretation with statistical language models to create a model of correct programs. We train on a large codebase by extracting sequences of properties, which we call sentences. A student’s submission is then scored by abstracting it into sentences, which are ranked according to the language model. We devised a set of rules that can be applied in order to synthesize a correct program. Our tool performs beam search to find the sequence of rule applications which improve the submission’s score the most.

Our results indicate that in over two thirds of all cases, the corrected version of a program scores higher than the original submission. Furthermore, they show that our tool can find good fixes for programs with an an accuracy of 0.63.
Acknowledgment

I would like to thank Professor Martin Vechev for giving me the opportunity to work on such an interesting project. Despite his busy schedule, he always found time to discuss the next steps to take and new ideas to explore.

I would also like to express my thanks to Veselin Raychev, who always found time to help me with problems, even when he was in a different timezone.

Finally I would like to thank all my friends who supported me during the last six months, whether in talking ideas through, proofreading my thesis, or simply staring at me until I got my work done.
# Contents

List of Figures ........................................ vii
List of Tables .......................................... ix
List of Algorithms ...................................... xi

1. Introduction ........................................ 1
   1.1. Related Work .................................... 2
   1.2. Structure of the Thesis .......................... 3

2. Background .......................................... 5
   2.1. Abstract Interpretation .......................... 5
   2.2. Language Models .................................. 6

3. Abstraction .......................................... 9
   3.1. Concrete Semantics ............................... 9
   3.2. Instrumented Concrete Semantics ............... 10
       3.2.1. Per Object Semantics ....................... 10
       3.2.2. Dataflow Style Semantics ................. 11
   3.3. Abstract Semantics ............................... 14
   3.4. Tweaking the semantics .......................... 15

4. Scoring ............................................ 17
   4.1. Computing the Score ............................. 17
   4.2. Capabilities and Limitations of the Score .... 18
List of Figures

3.1. Sentences obtained using the instrumented per object semantics. 11
3.2. Sentences obtained using the instrumented dataflow style semantics. 13
3.3. Two programs where the variable \( a \) is written in different cases. 14
3.4. Sentences resulting from the programs in Figure 3.3a and 3.3b. 14

4.1. Illustration of score which separates good from bad programs. 17
4.2. Two programs which are contrary to their behaviour. 19

5.1. Illustration of trying to find the best program in a program’s neighbourhood. 21
5.2. Illustration of the Return Rule returning any local variable. 23
5.3. Example for the application of the Return Rule to a program. 23
5.4. Example for the application of the Loop Iterate Rule to a program. 23
5.5. Example for the application of the Remove Functions rule to a program. 24
5.6. Example for the application of the Insert Test Rule to a program. 25
5.7. Example for the application of the Invert Conditions Rule to a program. 25
5.8. Example for the application of the Modify Comparison Rule to a program. 26
5.9. Potential search tree for applying rules. The arrows denote that one rule is applied, the resulting program with the best score is marked yellow and the final best ranked program is marked orange. 27

6.1. General overview over the architecture of our tool. 29

7.1. Overview over the distribution of sentence length for both abstractions. 38
7.2. Comparison of average score for correct programs (orange) and incorrect programs (blue) to the task `computeDeriv`. 38
7.3. Example of a fix under dataflow style abstraction which did not maintain the coding style of the student. 40
List of Figures

7.4. Example of a fix under dataflow style abstraction which introduces more sentences, which lowers the score. ........................................... 41
7.5. Example of a fix under per object abstraction which does not reflect as change in the sentences, and thus leads to the identical score. ................. 41
List of Tables

7.1. Comparison of the score of programs fixed in the user study under dataflow style abstraction. ................................................................. 39
7.2. Comparison of the score of programs fixed in the user study under per object abstraction. ................................................................. 39
7.3. Results from running the tool in MODIFY mode under the dataflow abstraction. 42
7.4. Number of applications for each rule. ................................................................. 43
List of Tables
List of Algorithms

6.1. Algorithm to lazily compute the depends-on relation while evaluating statements. 32
6.2. Helper function to evaluate the expressions in Algorithm 6.1. . . . . . . . . . . . . 33
6.3. Algorithm to apply rules to AST. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
List of Algorithms
Introduction

Massive open online courses are very popular. Platforms like coursera\(^1\) or edx\(^2\) offer a wide variety of courses on computer science topics and people from all over the world participate. Organizing such a course is a lot of effort, with a significant portion of the time spent in evaluating and grading student submissions of programming assignments. Course organizers usually set up automated test suites, but the feedback provided by such "judges" often lacks detailed information. Instead, it just present a "correct" or "wrong answer" judgment to the student. Such feedback can be frustrating, especially to novice programmers.

In this work, we propose a novel approach for automatically fixing incorrect programs. Our goal is to provide the student with individual instructions stating what needs to be changed in order to obtain a better program. We achieve this through a combination of program analysis, statistical language modeling, and program synthesis, with the main focus on building a model of correct programs using a large amount of existing code from open source projects. Each student submission is then ranked with our model and a fixed version of the submission is produced, which improves in score compared to the original program. The student will then receive the changes necessary to construct the fixed version as a feedback. Programmers with small mistakes in their algorithm will benefit from these detailed suggestions, rather than a simple "wrong answer".

To be more exact, we use a probabilistic model based on statistical language models to describe what correct programs look like. To this end, we extract program properties using static analysis and combine them into sentences. These sentences are used to build the language model. We then can query the model using the sentences from a submission and obtain a score. The synthesizer then applies a set of modifications to the submission and re-evaluates the modified

\(^{1}\)https://www.coursera.org/  
\(^{2}\)https://www.edx.org/
1. Introduction

programs. The modifications which result in the best score can the be checked for correctness and presented to the student.

Our contributions are therefore the following:

- We present an automated technique for ranking programs, for which incorrect programs have a lower score than correct problems solving the same problem. Our ranking technique operates without knowledge of the specific problem the program tries to solve.
- We propose a synthesis procedure that performs modifications which improve the score of a program. We show that such an approach is useful both for fixing incorrect programs as well as for improving the coding style.
- We propose an automated tool that creates individual feedback for each submission. Our tool does not rely on any specification or test suite, therefore it can be used for any task.

1.1. Related Work

There have been multiple attempts at automatically fixing incorrect programs. Douce et al. [Douce et al. 2005] provide an overview of the history of automatic test-based assessment of a program. They acknowledge that it is difficult to manage and correct submissions by a large number of students, and address existing tools and systems to assess the assignments. This includes different grader programs which run test cases against the submission, tools that analyse for more complex criteria such as CPU time and style, and the necessary means to administrate and review submissions. However, all assessment systems discussed are in need of a test suite for each task that students have to solve, and staff to administer and often also manually review submissions. The tool developed in this thesis does not need a test suite and although we do not manually review the submissions, we can provide the student with richer feedback than just "program OK" or "wrong answer". This distinguishes us from the tools mentioned by Douce et al.

Singh et al. [Singh et al. 2013] present Autograder, an approach at providing more helpful feedback to the student. They rely on the complete specification as well as a reference solution for the task the programs have to solve, as well as a set of errors that they expect people to make while solving this task. Using this information, they rewrite the program to contain certain constraints and feed it into a synthesizer, from which they draw output to give the students. We use the same idea to obtain rules for common mistakes, however, our approach does neither rely on the specification nor on common errors for this task. Thus, we can attempt to correct programs we have never seen before, while Autograder requires a reference implementation.

The combination of program analysis and statistical models is already used by Raychev et al. [Raychev et al. 2014] to perform code completion. They apply static analysis on a program with holes in it to extract sentences of method calls, then feed them into the language model and synthesize a fix based on the resulting probabilities. Our first abstraction is derived from their approach, but we consider much more than just method calls. Also, while the general idea is the same, we use a different synthesis procedure and only operate on complete programs.
1.2. Structure of the Thesis

The rest of this thesis is organized as follows: Chapter 2 will provide some background on program analysis and statistical language models. Chapter 3 presents the details of the abstraction used to represent a program as a set of sentences. The details of the language model are discussed in Chapter 4, and the synthesis procedure and various rules are shown in Chapter 5. Finally, we will detail our implementation of the tool in Chapter 6 and present the evaluation of it in Chapter 7. Chapter 8 concludes the thesis with an outlook on future work.
1. Introduction
Background

In this chapter, we will give a short overview of the necessary background for this thesis. We rely on two well studied concepts: abstract interpretation and language models. Abstract interpretation is used in our static analysis, and the statistical language model is used to obtain a score for a program.

2.1. Abstract Interpretation

Abstract interpretation is a widely used technique in static analysis to approximate the behaviour of a program without or by only partially executing it. It works over two domains, the concrete domain and the abstract domain. The concrete domain consists of possible executions of a program in all possible environments. However, arguing about certain program properties in the concrete is often not feasible, therefore one has to rely on abstraction. An abstraction of a program or part of a program results in an approximation of the program behaviour, which enables us to examine specific properties, but may inhibit us in discussing others.

As a short example, consider the following function:

```python
function m(x):
    if x >= 0:
        y = 2*x + 1
    else:
        y = -2*x + 1
    return y
```

Say we are interested in knowing if the result of \( m(x) \) is even or odd. If we want to discuss this in the concrete domain, we would have to enumerate all cases of \( x \) and compute the result. But
if we drop some information, e.g. the value of \( x \), and use the simple concept of numbers just being "odd" or "even", we can easily argue that no matter if \( x \) is even or odd, we will always end up with an odd number: In both branches, \( x \) is multiplied by an even number, which will result in an even number again. Adding one to it will make the number odd. Thus, both branches result in an odd number. This is an example of using an abstraction.

However, in the "odd-even-domain", we cannot argue about whether or not the result of \( m(x) \) will be greater than zero. We would need a different domain for this, e.g. the Interval Domain. As we can see, we have to drop some information to be able to reason about certain properties, but we will not be able to maintain reasoning about other properties due to the information loss.

One way to define an abstraction is to start out with *concrete semantics*, which describe the program’s behaviour in the concrete domain and the so called *state* of a program at a certain execution point. If we want to track specific properties in the concrete domain, we add them to the *state*, thus creating *instrumented concrete semantics*. From these, we can then find an abstraction of the behaviour to acquire the *abstract semantics*, using which we can reason about the properties we are interested in.

In order to prove a property, we evaluate each statement of a program under the abstraction. This will give us a state for each statement. From the state, we can read out if the property holds or not.

### 2.2. Language Models

Language models are used in many applications of natural language processing, such as speech recognition or machine translation. There are many different techniques, e.g. regular expressions for relaxing search terms, finite state automata to recognize very simple languages, and also statistical models to predict how a sentence in a language will continue. For this thesis, we use a specific kind of statistical model, so we will focus on this model.

Statistical language models are often used for word prediction, which goes in hand with the task of estimating the probability of a sentence. Consider the *vocabulary* \( V \) of your language and a sentence \( w_1 w_2 \cdots w_n \) composed of words \( w_i \in V \). Language models try to compute the probability \( P(w_1, w_2, \cdots, w_n) \), that is, the joint probability that each word occurs in its current position. The higher the probability, the more likely the sentence is a correct sentence of that language. But how is this probability computed?

The joint probability can be written as

\[
P(w_1, w_2, \ldots, w_n) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1 w_2) \cdots \cdots P(w_n|w_1 w_2 \cdots w_{n-1})
\]

However, we still do not know how to compute the probability \( P(w_i|w_1 \cdots w_{i-1}) \), that is, the probability that given a sequence of preceding words \( w_1 \cdots w_{i-1} \) the next word is \( w_i \). The solution to this is to approximate these probabilities by assuming that the probability of a word depends only on the \( n - 1 \) previous words, leading to the \( n \)-gram model.

A \( n \)-gram is simply a subsentence of length \( n \). The simplest form of an \( n \)-gram model is the *unigram* model, where each \( n \)-gram consists of one word only. This model has the assumption
that all words are independent of each other:

\[ P(w_1, w_2, \ldots, w_n) \approx \prod_{i=1}^{n} P(w_i) \]

However, this model may not be very useful, as it will prefer frequent words (such as "the") over infrequent words (e.g. "fox"), even when the context suggests a less frequent word. As an example, the sentence "the quick brown ___" calls for the word "fox" instead of the much more frequent word "the".

The bigram model will approximate the probability for a word given the previous part of the sentence by only looking at the last word. Thus, it will approximate \( P(\text{fox}|\text{the quick brown}) \) with \( P(\text{fox}|\text{brown}) \). Inserting an artificial "start of sentence"-character \(<s>\) as \( w_0 \), we compute the approximated probability as follows:

\[ P(w_1, w_2, \ldots, w_n) \approx \prod_{i=1}^{n} P(w_i|w_{i-1}) \]

Similarly, a trigram model always considers the last two words, and in general, an \( n \)-gram model will always consider the last \( n - 1 \) words. This makes it a Markov model. We train the model by counting the \( n \)-grams in the training corpus.

As \( n \)-gram models work by multiplying probabilities, which, by definition, are less than one, the result will be a very small number. Thus, in order to prevent arithmetical underflow, it is more common to work on the logarithm of the probability, the \( \logProb \). Because normal multiplication corresponds to addition in logarithmic space, we are less prone to numerical problems.

\( n \)-gram models have the limitation that they highly depend on their training corpus. If they suffer from data sparseness, that is, the sentence we want to estimate contains a lot of words which are out of vocabulary (OOV), which means they never show up in the corpus, the model needs to apply smoothing. Smoothing attempts to re-estimate the counts, such that an estimate for unknown words can be gained. There are different smoothing algorithms, e.g. [Good 1953], [Witten and Bell 1991] or [Kneser and Ney 1995] and the modified version of Kneser-Ney by [Chen and Goodman 1999]. They all follow the idea to use the count of words which appear once to estimate the count of an unknown word.

For further details on the theory behind \( n \)-grams or the different smoothing mechanisms, refer to [Jurafsky and Martin 2004, Ch. 4].
2. Background
Abstraction

In this chapter, we will discuss the first step of our approach at automatically assessing a program: the abstraction. We will give a formal description of all the representations we use and an overview of possible parameters to tune them. Furthermore, we will discuss the problems each representation has.

3.1. Concrete Semantics

We use standard concrete semantics, defined as follows. Let $\text{objects}^\sharp$ denote the unbounded set of dynamically allocated objects as well as primitive types and constants such as numbers and string constants. Then:

$\mathcal{L}^\sharp \in \mathcal{P}(\text{objects}^\sharp)$

$v^\sharp \in \text{Val} = \text{objects}^\sharp \cup \{\text{null}\}$

$\rho^\sharp \in \text{Env} = \text{VarIds} \rightarrow \text{Val}$

$h^\sharp \in \text{Heap} = \text{objects}^\sharp \times \text{FieldId} \rightarrow \text{Val}$

$\text{state}^\sharp = \langle \mathcal{L}^\sharp, \rho^\sharp, h^\sharp \rangle \in \text{States} = \mathcal{P}(\text{objects}^\sharp) \times \text{Env} \times \text{Heap}$

In other words, a program state keeps track of the allocated objects as well as a mapping of local variable ids to values and a mapping of fields of allocated objects to values.

When evaluating a statement, we assume standard interpretation for a transition from a state $\langle \mathcal{L}^\sharp, \rho^\sharp, h^\sharp \rangle$ to a newly generated state $\langle \mathcal{L}'^\sharp, \rho'^\sharp, h'^\sharp \rangle$. 
3. Abstraction

3.2. Instrumented Concrete Semantics

We now show how we instrument these concrete semantics to suit our approach. We developed two different ideas and will describe a way how to obtain the interesting properties for each of them as well as the limitations these approaches have.

3.2.1. Per Object Semantics

We are interested in seeing which operations are carried out on an individual object, thus we augment our concrete semantics to track what happened to each object. We define a mapping from each object to a sentence, which is formed by a sequence of events. An event for an object o corresponds to an expression containing o, where o can be either an argument to the expression or the result of it. Examples for events are binary operations, comparisons, and method invocations. This means, formally, that an event is a pair $⟨e, pos⟩$ where e denotes the considered expression, e.g. Add for an addition operation or format for a call to a method called "format", and pos denotes the position of the object o in the expression. For the case where o is the result of the expression, we have a specific ret position.

Take as an example the addition expression $y + z$. This expression would result in the three events $y : ⟨Add, 0⟩$, $z : ⟨Add, 1⟩$, and $o_{ret} : ⟨Add, ret⟩$.

A sentence for the object o is formed by the sequence of events on o. An object can be mapped to the empty sentence $ε$, which means the program contains no expressions involving this object. We denote the set of all sentences by $S$.

We retrieve our instrumented semantics by augmenting the state $⟨L^k, ρ^k, h^k⟩$ with a mapping $seqs^k : L^k → S$, which maps each concrete object to its sentence.

For a given state $⟨L^k, ρ^k, h^k, seqs^k⟩$, we obtain a new state $⟨L'^k, ρ'^k, h'^k, seqs'^k⟩$ by applying standard interpretation for $L^k$, $ρ^k$, and $h^k$ when evaluating a statement. To update $seqs^k$, we use the transition function $⟦[ stmt ]⟧^k : statement × seqs^k → seqs'^k$, which takes a statement $stmt$ and a mapping of sentences $seqs^k$ and produces an updated mapping of sentences $seqs'^k$. For readability, we use shorthand notion $⟦[ stmt ]⟧^k_{seqs^k}$ instead of $⟦[ stmt ]⟧^k_{seqs^k}$. Furthermore, let $e_i$ be an expression. An expression can either be a terminal, such as a variable identifier (e.g. $x$) or a binary operation involving two expressions (e.g. $e_1 + e_2$). We use $⊕$ to denote an arbitrary operation on expressions. For simplicity, we only show binary operations here, although this can easily be extended to n-ary operations. Assume that operator precedence is obeyed. The transition function works as follows:

- $[x := new T()]^k_{seqs^k}$ will allocate a new object $o_{new} ∈ objects^k \setminus L^k$, which we map to the empty sentence: $seqs^k(o_{new}) = ε$. From standard interpretation, we get $ρ^k(x) = o_{new}$.

- $[r := e_1 ⊕ e_2]^k_{seqs^k}$ will extend the sentence of the objects pointed to by $e_1$, $e_2$, and $r$ by the event $⊕$. In detail, let $o_1 = ρ^k(e_1)$, $o_2 = ρ^k(e_2)$, and $o_{ret} = ρ^k(r)$. Then, for $e_1 ⊕ e_2$, we will extend the sentences the following way:

  $$seqs'^k(o_i) = seqs^k(o_i) · ⟨⊕, i⟩$$

  with $i ∈ 1, 2$

  Furthermore, we will extend the sentence of $o_{ret}$ with the event $⟨⊕, ret⟩$. 


3.2. Instrumented Concrete Semantics

```python
def getAst(file):
    code = open(file).read()
    t = ast.parse(code)
    return t
```

(a) Code snippet. (b) Sentences for code snippet.

Figure 3.1: Sentences obtained using the instrumented per object semantics.

An example of such sentence mappings can be seen in Figure 3.1. For method calls, the position 0 indicates that this object is the receiver of the method.

Limitations

Sentences which are created with this semantics tend to be rather short. The reason is that a lot of dynamic objects are created when evaluating an expression. For example, a comparison will return a new boolean object indicating the result of the comparison. Unfortunately, there are many sentences containing only two words: the event in which this object was returned and the event in which the object was used next. Examples for this are the objects `obj1` and `code` in Figure 3.1. However, as input for our language model, we prefer longer sentences which reflect more of a program’s behaviour. The model operates on counting the \( n \)-grams of the training data and computing probabilities based on the \( n \)-grams of the test data. The shorter these sentences, the more inaccurate the probability estimation will get, especially if the typical sentence is shorter than \( n \) words.

3.2.2. Dataflow Style Semantics

The second instrumented semantics we developed is slightly more complex. We want to overcome the shortcoming of the per object semantics, which causes the scattering into many objects and the short sentences generated. Thus we focus on dependencies between the objects. As a motivating example, take the following piece of code:

```python
for i in range(len(l)): # l = some list
    ...
```

Following the per object semantics, we get the following sentences:

\[
l \rightarrow \langle \text{len}, 1 \rangle
\]

\[
obj_1 \rightarrow \langle \text{len}, \text{ret} \rangle \langle \text{range}, 1 \rangle
\]

\[
obj_2 \rightarrow \langle \text{range}, \text{ret} \rangle \langle \text{In}, - \rangle \ldots
\]

etc.
3. Abstraction

We omitted the rest of the sentences, as we already discovered that there obviously is a connection between \( l \) and the result object of the call to \( \text{len}(l) \) (= \( \text{obj}_1 \)), which we would like to be represented in our instrumented semantics. We can say that \( \text{obj}_1 \) depends on \( l \). To reflect this, we decided to append the sentence of \( \text{obj}_1 \) to the sentence of \( l \). Similarly, we can say that \( \text{obj}_2 \) depends on \( \text{obj}_1 \), as it is the result of an operation being applied to \( \text{obj}_1 \). If we "glue" all these sentences together, we obtain something that almost represents the flow of an object through the program:

\[
l \rightarrow \langle \text{len}, 1 \rangle \langle \text{range}, 1 \rangle \langle \text{In}, - \rangle \ldots
\]

Let us formalize this the following way: We define a depends-on relation

\[
\prec \subseteq \text{objects}^s \times \text{objects}^s
\]

For two objects \( \text{obj}_1 \) and \( \text{obj}_2 \), the relation holds if \( \text{obj}_1 \) depends on \( \text{obj}_2 \), that is, if \( \text{obj}_1 \) is the result of an expression containing \( \text{obj}_2 \). For example, in \( x = a + b \), we will have \( x \prec a \) and \( x \prec b \). The depends-on relation is reflexive and transitively reduced.

Moreover, we introduce a parameter \( k \), which denotes the dependency level. That is, \( \text{obj}_1 \prec^k \text{obj}_2 \) indicates that within maximum \( k \) steps, one can reach \( \text{obj}_2 \) from \( \text{obj}_1 \) in the depends-on relation.

As we want to track the flow of objects through the whole program, we use the transitive closure of the depends-on relation, denoted by \( \prec^* \).

The depends-on relation is global and can be easily precomputed. Thus, it is not part of our state.

We can now use the \( \prec \) relation for dependency tracking. We augment our concrete semantics to contain a mapping of concrete objects to their sentences, where a sentence, as well as an event and the set of sentences, is defined as in Section 3.2.1. However, we omit the event of being returned and therefore do not have a special position for \( \text{ret} \). To reflect the dependency between objects, we now allow each object to map to the sentences of all objects it depends on and all objects which depend on it. We obtain the instrumented semantics by augmenting the state \( \langle L^x, \rho^x, h^x \rangle \) with a mapping \( \text{seqs}^x \), which maps now each object to a set of sentences consisting of the union of all sentences of all its dependent objects and all objects it depends on.

Thus, \( \text{seqs}^x : L^x \rightarrow \mathcal{P}(S) \).

This means, that if we update an object \( o \), the update will propagate to all objects \( o' \) which depend on \( o \), as well as all objects \( o' \) on which \( o \) depends. That is, all objects \( o' \) with \( o \prec^* o' \) or \( o' \prec^* o \) are updated. We will discuss this later in an example.

Upon evaluating a statement, the given state \( \langle L^x, \rho^x, h^x, \text{seqs}^x \rangle \) is changed to the new state \( \langle L^x, \rho^x, h^x, \text{seqs}^x \rangle \) by applying standard interpretation for \( L^x, \rho^x, \) and \( h^x \). The \( \text{seqs}^x \) component is again updated using a transition function \( \llbracket \text{Statement} \rrbracket : \text{Statement} \times \text{seqs}^x \rightarrow \text{seqs}^x \) with the same definitions and assumptions as in Section 3.2.1. The transition function works as follows:

- \( \llbracket x := \text{new} \ T() \rrbracket \text{seqs}^x \) will allocate a new object \( o_{\text{new}} \in \text{objects}^x \setminus L^x \), which we map to the empty sentence: \( \text{seqs}^x(o_{\text{new}}) = \epsilon \). From standard interpretation, we get \( \rho^x(x) = o_{\text{new}} \).
- \( \llbracket r := e_1 \oplus e_2 \rrbracket \text{seqs}^x \) will extend the sentence of the objects pointed to by \( e_1 \) and \( e_2 \) by the event \( \oplus \). In detail, let \( o_1 = \rho^x(e_1) \), \( o_2 = \rho^x(e_2) \) and \( o_{\text{rel}} = \rho^x(r) \). Then, for \( e_1 \oplus e_2 \), we will extend the sentences the following way:

\[
\forall o_j, o_j \prec^* o_i \text{ or } o_i \prec^* o_j : \text{seqs}^x(o_j) = \{ \text{seq} \cdot (\oplus, i) | \text{seq} \in \text{seqs}^x(o_j) \}
\]
3.2. Instrumented Concrete Semantics

def getAst(file):
    code = open(file).read()
    t = ast.parse(code)
    return t

(a) Code snippet.

code = open(file).read()

file → { ⟨open, 1⟩ ⟨read, 0⟩ }
code → { ⟨open, 1⟩ ⟨read, 0⟩ }

\[ t = ast.parse(code) \]  

file → { ⟨open, 1⟩ ⟨read, 0⟩ ⟨parse, 1⟩ }
code → { ⟨open, 1⟩ ⟨read, 0⟩ ⟨parse, 1⟩ }
ast → { ⟨parse, 0⟩ }
t → { ⟨open, 1⟩ ⟨read, 0⟩ ⟨parse, 1⟩ , ⟨parse, 0⟩ }

return t

file → { ⟨open, 1⟩ ⟨read, 0⟩ ⟨parse, 1⟩ ⟨Return, −⟩ }
code → { ⟨open, 1⟩ ⟨read, 0⟩ ⟨parse, 1⟩ ⟨Return, −⟩ }
ast → { ⟨parse, 0⟩ ⟨Return, −⟩ }
t → { ⟨open, 1⟩ ⟨read, 0⟩ ⟨parse, 1⟩ ⟨Return, −⟩ , ⟨parse, 0⟩ ⟨Return, −⟩ }

(b) Sentences for code snippet, shown step by step.

Figure 3.2: Sentences obtained using the instrumented dataflow style semantics.

Furthermore, we will update the sentences of \( o_{ret} \) according to the dependency information:

\[
seqs^{\gamma'}(o_{ret}) = \{ seqs^{\gamma}(o_j) \mid o_j \prec^* o_{ret} \} \cup \{ seqs^{\gamma}(o_j) \mid o_{ret} \prec^* o_j \}
\]

An example of such sequence mappings can be seen in Figure 3.2. Note that the code snippet is exactly the same as in Figure 3.1. There are fewer sentences and the sentences themselves are longer, providing more insight in the sequence of operations that were applied to the objects.

Limitations

There are a few limitations to the dataflow style abstraction. First, for an event \( e \) in a sentence \( s \) of an object \( obj \) obtained through the dataflow style semantics, one usually cannot tell if \( e \) originated from an operation applied on \( obj \) or on any object depending on \( obj \) or \( obj \) depending on it. As an example, consider the following code:

```python
def m(j):
    a = j + 1
    b = _ * 2 # insert here either j or a
```

No matter whether we insert \( j \) or \( a \) at the underscore, the event \( \langle Mult, 0 \rangle \) will be added to the sequences of both \( j \) and \( a \), as they depend on each other. However, only one of the two objects was actually involved in the operation. In not being able to differentiate the objects, we actually lose information.
3. Abstraction

\[
\begin{align*}
\text{if } (x < y) : & \quad \text{if } (x \geq y) : \\
da & = b + 4 & a & = b + 4
\end{align*}
\]

(a) \hspace{2cm} (b)

*Figure 3.3.* Two programs where the variable \(a\) is written in different cases.

\[
\begin{align*}
x & \rightarrow \{(Lt, 0), (GtE, 0)\} \\
y & \rightarrow \{(Lt, 1), (GtE, 1)\} \\
b & \rightarrow \{(Add, 0)\} \\
4 & \rightarrow \{(Add, 1)\} \\
a & \rightarrow \{(Add, 0), (Add, 1)\}
\end{align*}
\]

*Figure 3.4.* Sentences resulting from the programs in Figure 3.3a and 3.3b.

Another drawback is that to a certain degree, we do still not track enough dependencies, especially when considering control flow. If the condition of an if-statement does not use objects which appear in one of the branches following the test, the sentences generated for these objects will become flow-insensitive. As an example, consider the code in Figure 3.3a, which will result in the sentences displayed in Figure 3.4. These sentences are however identical to the ones obtained from the program in Figure 3.3b, although the condition is inverted, and thus, the semantics of the program have changed.

### 3.3. Abstract Semantics

As mentioned in Section 2.1, once the concrete semantics have been defined, we have to set up the abstract semantics. We need them, because computing the sentences under the instrumented semantics is generally not feasible. The following semantics are common to the abstractions of both instrumented concrete semantics.

First, we define the notion of an *abstract object* and an *abstract sentence*. We use a very basic flow insensitive points-to analysis which partitions the set \(\text{objects}^5\) into a set of *abstract objects* denoted by \(\text{objects}\). An *abstract sentence* for an abstract object \(\text{obj}_a\) corresponds to a concrete sentence of finite length. As we have to account for control flow through the program, each abstract object \(\text{obj}_a\) is mapped to a set of *abstract sentences*: \(\text{seqs} : L \rightarrow \mathcal{P}(\mathcal{S})\). To ensure the sentences are of finite length, we bound the number of loop iterations through loop unrolling.

With these details, our instrumented abstract program state is \(\langle L, \rho, h, \text{seqs} \rangle\). The abstract semantics for evaluating a statement differs between the two abstractions. The abstract dataflow style semantics follow the instrumented concrete dataflow style semantics with the exception that they consider an abstract transformation function \(\mu\), abstract objects, and sets of abstract sentences instead of the concrete transformation function, concrete objects, and concrete sentences.

The abstract per object semantics change as follows:

- \(x := \text{new } T()\) \(_{\text{seqs}}\) will get a new abstract object \(a_{\text{new}} \in \text{objects}\), to which we map a set
3.4. Tweaking the semantics

containing the empty sentence: \( \text{seqs}'(a_{new}) = \text{seqs}(a_{new}) \cup \{\epsilon\} \).

- \( [r := e_1 \oplus e_2] \text{seqs} \) will extend each sentence of the abstract objects \( a_1 = \rho(e_1), a_2 = \rho(e_2) \), and \( a_{ret} = \rho(r) \) by the event \( \oplus \), the following way:
  \[
  \text{seqs}'(a_i) = \text{seqs}(a_i) \cdot (\oplus, i) \quad \text{with} \ i \in \{1, 2, \text{ret}\}
  \]

As we have bounded the domain of abstract sentences to only include sentences of finite length, we can guarantee the analysis to reach a fixed point. In practice, it may still take a long time and result in exponentially many sentences in each abstract sentence, because of the control flow. To overcome this problem, we limit the number of branches we follow. Additionally, we offer the possibility to limit the number of sentences collected per object.

3.4. Tweaking the semantics

The two semantics we defined in Sections 3.1 to 3.3 can be further parametrized to allow for a wider range of abstractions under the same concrete semantics. This could be useful to tune the abstraction exactly to the data to maximize efficiency. We can introduce the following "knobs" to parametrize this abstraction:

- **Include or exclude an object**: This parameter aims at focusing the abstraction on the objects that matter for our desired property. We can easily omit events connected to an object and ignore expressions containing an excluded object. At the same time, we can decide to track only certain objects and all expressions they are involved in.

- **Exclude a statement**: Similar to the exclusion of an object, we can decide not to track events which result from the evaluation of a specific statement. For example, we can limit the abstraction to a critical section of the program and omit the set-up and teardown code around this section from analysis. This will again help in tuning the abstraction to more relevant parts of the program.

- **Adjust dependency level \( k \)**: We can decide how much dependency between different objects we want to track, ranging from no dependency information (per object semantics) over only directly dependant objects to fully transitive dependency tracking, as is done in the dataflow style abstraction, by varying the propagation level when applying updates. This is done through specifying the dependency level \( k \) and using \( \prec^k \) instead of \( \prec^* \) in the concrete dataflow style semantics.

- **Reorder dependency evaluation**: When appending an event in the dataflow style semantics, it makes a difference if we first append to \( \text{seqs}^2(o_1) \) or \( \text{seqs}^2(o_2) \), as it is possible that they share objects they depend on. We can vary the evaluation order to see if we can gain more information from it.

For this thesis, we did not implement and evaluate the knobs, they are left as future work.
3. Abstraction
Scoring

Once we obtained the set of sentences for a program, we are interested in obtaining a score for them. The score should express how "good" or "bad" a program is. The higher the score, the better the program. Ideally, all good programs have a high score and all bad ones are scored low, so our original intent was to be able to separate the correct, "good" programs from the incorrect "bad" programs. This is illustrated in Figure 4.1.

This chapter will focus in how to compute the scores, what they signify, and what limitations we have to deal with.

4.1. Computing the Score

The score is computed using a \( n \)-gram language model over the sentences obtained through the abstractions from Chapter 3. Recall from Section 2.2 that in statistical language models, we are interested in computing a probability for a sentence \( w_1 w_2 \cdots w_n \) composed of words \( w_i \in V \), where \( V \) is some vocabulary. This probability \( P(w_1, w_2, \ldots, w_n) \) can also be computed as a

\[ P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i | w_1, w_2, \ldots, w_{i-1}) \]

**Figure 4.1.: Illustration of score which separates good from bad programs.**
4. Scoring

joint probability

\[ P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i \mid w_1 \cdots w_{i-1}) \]

Since we do not know the conditional probability \( P(w_i \mid w_1 \cdots w_{i-1}) \), we approximate it using \( n \)-grams.

We train our language model by extracting the sentences from a corpus of programs. Using a Kneser-Ney discounting algorithm, we construct a \( n \)-gram model from these sentences. Note that as we are only interested in which sentences can be generated from this corpus, we discard any information relating to the objects which map to the sentences.

After building the model, we can use it to score a program. Because our analysis is intraprocedural, we assume all programs to only contain one method. However, the score could easily be extended to multiple methods per program. Given a program \( Prog \), we abstract it in order to extract the set of sentences \( seqs_{Prog} \), which contains the sentences of all objects of \( Prog \). Again, we are not interested in which object the sentences belong to, so we discard this information.

But we cannot just use the probability. Suppose you have sentences of different length, e.g. the following two which were obtained using dataflow style abstraction:

- \( \langle Add, 0 \rangle \langle Add, 0 \rangle \langle Print, 0 \rangle \)
- \( \langle range, 1 \rangle \langle In, - \rangle \langle SubscriptAccess, 1 \rangle \langle Mult, 0 \rangle \langle Pow, 1 \rangle \langle Mult, 1 \rangle \langle Add, 1 \rangle \langle Print, 0 \rangle \)

The probability for the first one is \(-4.78164\) and the second one has probability \(-11.6252\). However, the second sentence consists of more words than the first one. Usually a sentence with more words will have a smaller probability, as to compute it there will be more multiplications with numbers less than one. To make the sentences comparable, we normalize the probability by the number of words per sentence. This results in a normalized probability of \(-1.59388\) for the former sentence and \(-1.45315\) for the latter. Evidently, this enables us to actually compare the two sentences and discover that the longer sentence is more likely for the language.

From the set of normalized probabilities \( probs_{Prog} \) for all sentences in \( seqs_{Prog} \), we now compute a score for the whole program \( Prog \). In the beginning, we took the minimal normalized probability \( score_{min} \) with \( score_{min} = \min_{p_s} p_s \) for any \( p_s \in probs_{Prog} \). The intention was that \( score_{min} \) should correspond to the worst trace possible. But we soon discovered that many programs, good and bad, shared this trace, so there was no point in basing the score on it. We then moved to taking the average of all normalized probabilities, \( score_{avg} \). This average is our final score for the program.

4.2. Capabilities and Limitations of the Score

When scoring programs, being "bad" does not necessarily mean incorrect behaviour, it could also indicate a correct program with a very poor coding style. At the same time, "good" can represent a program that has a correct algorithm, but a mistake in the implementation which will result in a runtime error. This is a capability of our scoring mechanism - we can rank programs with a better style higher than programs with a low style, even if the program is not completely
4.2. Capabilities and Limitations of the Score

def computeDeriv(poly):
    if (len(poly) < 2):
        return [0.0]
    deriv = poly[1:]
    i = 2
    while i < len(poly):
        deriv[(i-1)] *= i
        i += 1
    return deriv

(a) A correct program which is written in a bad style.

def computeDeriv(poly):
    if (len(poly) == 1):
        return [0.0]
    deriv = []
    for i in range(1, len(poly)): # <-- Typo here!
        deriv.append(i*poly[i])
    return deriv

(b) An algorithmically correct program which will not work because of a typo.

Figure 4.2.: Two programs which are contrary to their behaviour.

correct.

For examples, refer to Figure 4.2. The program from Figure 4.2a gets a comparatively low score although it is a correct program. This is due to its poor coding style. The program from Figure 4.2b achieves a high score, as its algorithm is perfectly correct. However, because it is an incorrect program, we would prefer if it ranked lower.

There is an intuitive explanation for this. The score is based on the abstraction and on statistics. If the model is trained on a corpus of programs with similar coding style, it will have incorporated this style in its n-grams. If we now want to score a program with a very different coding style, such as the program from Figure 4.2a, which is reflected in the sentences and the n-grams, it will receive a lower score.

At the same time, for the program in Figure 4.2b, if we have an abstraction, this means we lose some information. It is in general possible that the abstraction does not capture the error of a program. Thus, our sentences will not reflect the mistake the programmer has made and the score for the program will be good, even though the program itself has incorrect behaviour. Notice that this is a limitation of the abstraction, not the approach itself.

Our main limitation is that we cannot just compare two random programs to each other and distinguish if one of them or both are incorrect. To overcome this, we want to condition on the problem the program attempts to solve. This lead us to a new hypothesis. For a given program $p$ and its score $score_{avg}$, we want to explore the programs surrounding it, that is, programs which are created by applying changes to $p$. If $p$ is a correct, good program, we want its score to remain better than the scores of the newly created programs. If $p$ is a bad program, we hope
4. Scoring

to find a better program in its surroundings, such that we can provide the user with feedback on how to change his program to improve it. These changes are discussed in the next chapter, Chapter 5.
Repair Rules

Our goal is to provide the user with feedback on how to improve his program so that it will be correct and match its task. We described an abstraction in Chapter 3 and a score in Chapter 4 to be able to rank programs among each other. Now, given a program, we want to search the space of programs which are a slight modification of the original program in order to find a better one. Figure 5.1 illustrates this. Our hypothesis is that the correct version of a program lies within a finite number of modifications away from the original program. In this chapter, we will focus on finding modification suggestions for a program, and ranking them. Sections 7.3 and 7.4 will discuss the experimental details of this approach.

5.1. Finding Rules

We want to obtain a set of rules to modify programs. These rules are of the form $S \rightarrow T$, where $S$ is a potentially partial source expression of the AST which is to be replaced by the expression

![Figure 5.1](image_url)

*Figure 5.1:* Illustration of trying to find the best program in a program’s neighbourhood.
5. Repair Rules

For example, we can have a rule for subscript accesses

\[ \text{var}_s[\text{var}_i] \rightarrow \text{some}_{-}\text{func}(\text{var}_s[\text{var}_i]) \]

that changes an occurrence of a subscript access \( l[i] \) to some function of this subscript access \( \text{some}_{-}\text{func}(l[i]) \). Note that the programs created by these rules need not be actual programs. It could be that we return a partial program. An example for this would be a rule

\[ \text{var}_x, \text{var}_y \rightarrow \text{var}_x \text{EXPR} \text{var}_y \]

This rule indicates to the user that the usage of a binary expression with these two variables is encouraged. Which binary expression exactly is up to him to decide. Similarly, we could have a rule where \( \text{EXPR} \) is concrete, such as

\[ \text{var}_x, \text{var}_y \rightarrow \text{var}_x + \text{var}_y \]

and decide to give the user only partial feedback, stating that we believe he wants to apply a binary expression, but omitting the actual operator \(+\).

As stated in Section 4.2, we originally aimed at being able to separate the programs into good and bad programs. We then wanted to find the rules from the characteristics that classify the programs into good and bad. However, for the reasons mentioned in Section 4.2, this approach did not work out.

Therefore, we draw the rules we use to modify programs from experience. As will be discussed in Chapter 7.3, we compare incorrect programs with their correct counterpart and derive what fixes we can commonly apply in order to arrive at the correct version. From these fixes, we construct rules. The following section will discuss each rule in detail.

5.2. The Rules

In general, there are three kinds of rules. The first kind are the correctness rules which capture general kinds of bugs. The second kind are correctness rules which are specific to a task. While they improve correctness for one task, they might not help in general. And the third kind of rules are the stylistic rules. They do not affect correctness but improve coding style. However, we expect that only very few rules are stylistic rules. All of the rules we found can be put into one of these categories.

At the same time, one can categorize the rules according to their update behaviour. We propose rules which insert expressions as well as rules which delete expressions and rules which replace expressions. In order to assess their effectiveness, we have implemented, applied and tested a rule of each of these categories.

Let us discuss the different rules now.

5.2.1. The Return Rule

This rule is straightforward. Whenever a program does not return anything, we add a `return` statement. You can return any of the declared variables, as illustrated in Figure 5.2. Addition-
5.2. The Rules

**5.2. The Rules**

```python
def dummymethod():
    x = 4
    y = 5
    return x
```

(a) A dummy method which does nothing.

**5.2. The Rules**

```python
def dummymethod():
    x = 4
    y = 5
    return y
```

(b) Apply the Return Rule and decide to return \( y \).

**5.2. The Rules**

```python
def oddTuples(aTup):
aTup[0:len(aTup):2] # <== \[\]
```

(a) Original code.

**5.2. The Rules**

```python
def oddTuples(aTup):
    return aTup[0:len(aTup):2]
```

(b) Code with rule applied.

### Figure 5.2: Illustration of the Return Rule returning any local variable.

ally, one can choose to return the result of an expression if the last statement of the function body was an expression. See Figure 5.3 for an example. The general form of this rule is

\[
\text{empty stmt} \rightarrow \text{return var}
\]

for any local variable \( \text{var} \) of this program.

This rule could be easily generalized to \( \text{empty stmt} \rightarrow \text{return EXPR(var*)} \), for a subset of all local variables \( \text{var*} \) which are given as an argument to the generic function \( \text{EXPR} \). The result would again be a partial program. However, the search space for this generalized rule might be too big to be feasible.

#### 5.2.2. The Loop Iterate Rule

Recall that our test data consists of student’s submissions to a massive open online course. A common mistake made was that the students did not find the correct iterate for a for-loop. Python supports iterating over a collection, but students were not aware on how to properly use it. Programs as in Figure 5.4 were quite common, thus we decided to introduce the following rule:

\[
\text{for \ var\ in \ var\ collection} \rightarrow \text{for \ var\ in \ range(len(var\ collection))}
\]

**5.2.2. The Loop Iterate Rule**

```python
def oddTuples(aTup):
    d = ()
    for i in aTup:     # <== \[\]
        if i % 2 == 0:
            d = d + (aTup[i],)
    return d
```

(a) Original code.

**5.2.2. The Loop Iterate Rule**

```python
def oddTuples(aTup):
    d = ()
    for i in range(len(aTup)):
        if i % 2 == 0:
            d = d + (aTup[i],)
    return d
```

(b) Code with rule applied.

### Figure 5.3: Example for the application of the Return Rule to a program.

**5.2.2. The Loop Iterate Rule**

```python
def oddTuples(aTup):
    d = ()
    for i in aTup:
        if i % 2 == 0:
            d = d + (aTup[i],)
    return d
```

(a) Original code.

**5.2.2. The Loop Iterate Rule**

```python
def oddTuples(aTup):
    d = ()
    for i in range(len(aTup)):
        if i % 2 == 0:
            d = d + (aTup[i],)
    return d
```

(b) Code with rule applied.

**5.2.2. The Loop Iterate Rule**

```python
def oddTuples(aTup):
    d = ()
    for i in aTup:
        if i % 2 == 0:
            d = d + (aTup[i],)
    return d
```

(a) Original code.

**5.2.2. The Loop Iterate Rule**

```python
def oddTuples(aTup):
    d = ()
    for i in range(len(aTup)):
        if i % 2 == 0:
            d = d + (aTup[i],)
    return d
```

(b) Code with rule applied.

**5.2.2. The Loop Iterate Rule**

```python
def oddTuples(aTup):
    d = ()
    for i in aTup:
        if i % 2 == 0:
            d = d + (aTup[i],)
    return d
```

(a) Original code.

**5.2.2. The Loop Iterate Rule**

```python
def oddTuples(aTup):
    d = ()
    for i in range(len(aTup)):
        if i % 2 == 0:
            d = d + (aTup[i],)
    return d
```

(b) Code with rule applied.

**Figure 5.4:** Example for the application of the Loop Iterate Rule to a program.
5. Repair Rules

An example of its application can be seen in Figure 5.4. This rule is Python-specific, however we are positive it can be adapted to other languages.

5.2.3. The Remove Functions Rule

Another mistake frequently made by students was the (sometimes incorrect) usage of functions when they are not necessary. This can be corrected by removing the call to the function. We turn this into a rule and give it the general form

\[ \text{some\_obj.some\_func(args\*)} \rightarrow \text{EXPR\_some\_func(some\_obj, args)} \]

For some function \text{some\_func}, this rule will remove the call to this function and replace it by a predefined expression which uses a subset of the arguments and the receiver. What expression it is and which arguments will be used depends on the instantiation of the rule.

Let us look at the instantiation for the Python built-in function \text{round(number[, ndigits])}. In the example in Figure 5.5, the students had the task of evaluating the value of a polynomial denoted by its coefficients at the point \( x \). The result is supposed to be a float. However, this call to \text{round()} will round the result to an integer, which is not the desired output type. For this reason, we derived the instantiation

\[ \text{round(number[, digits])} \rightarrow \text{number} \]

This will remove the call to \text{round()} and replace it with the un-rounded number. The rule itself is general, while each instantiation of it is task-specific.

5.2.4. The Insert Test Rule

Sometimes, students use a function right but they miss some guarantees that have to be met on the caller side. For example, a call to \text{list.remove(el)} will only succeed if the element \( el \) is still in the list. If it is not, then Python will issue a ValueError. In order to avoid this error, we need to introduce a test which ensures the element is present in the list. The general form of this rule is

\[ \text{obj.some\_func(args\*)} \rightarrow \text{i \notin TEST(obj, args\*) : obj.some\_func(args\*)} \]
5.2. The Rules

**Figure 5.6.** Example for the application of the Insert Test Rule to a program.

This rule is a general correctness rule because it will help prevent errors regardless of the task. However, in practice there needs to be a way to define which functions require a test and what condition the test has. This again limits the generality of the rule down to the specified locations.

### 5.2.5. The Invert Condition Rule

The invert condition rule is very straightforward. It is applied whenever someone has accidentally inverted a condition. The general form is

\[
\text{if } var_1 \text{ cond } var_2 \rightarrow \text{if } var_1 \text{ cond}^{\text{inv}} var_2
\]

and similarly

\[
\text{while } var_1 \text{ cond } var_2 \rightarrow \text{while } var_1 \text{ cond}^{\text{inv}} var_2.
\]

Figure 5.7 describes an example application of this rule. The student’s task was to enumerate all letters which are not contained in the list `lettersGuessed`. However, in his original code in Figure 5.7a, he accidentally enumerates only the letters in `lettersGuessed`. This rule will help fixing this case. It belongs to the replacing rules and is a general correction rule.

**Figure 5.7.** Example for the application of the Invert Conditions Rule to a program.
5. Repair Rules

5.2.6. Modify Comparison Rule

A very similar mistake to the inverted condition is mistaking $<$ for $\leq$ and $>$ for $\geq$ and vice versa. This can be generalized to the rules

\[
\text{if } \text{var}_1 \text{ comp } \text{var}_2 \rightarrow \text{if } \text{var}_1 \text{ comp}^\prime \text{ var}_2
\]

and

\[
\text{while } \text{var}_1 \text{ comp } \text{var}_2 \rightarrow \text{while } \text{var}_1 \text{ comp}^\prime \text{ var}_2
\]

where $(\text{comp, comp}^\prime) \in \{(\leq, \leq), (\leq, <), (\geq, >), (> , \geq)\}$. Again, this is a general correction rule and belongs to the category of replacing rules. An example for the application of this rule can be seen in Figure 5.8.

5.2.7. Further Rules

We could have easily presented more rules, like introducing an initialization statement for uninitialized variables, or correcting potential off by one errors, but in order to fix all programs, the list of rules needed would be infinitely long. Therefore, we do not devise more rules but try to see how far we get with the rules described here.

5.3. The Search Procedure

Once we obtain a set of rules, we use them to find a corrected version of a program $P^0$. To this end, we use the abstraction to extract a set of sentences $seqs_{P^0}$, from which we obtain a score $score_{P^0}$. Now we apply each rule separately, creating as many new programs $P^1_1, P^1_2, \ldots, P^1_n$ as there are locations to apply a rule. For each of these programs, we compute the score. For the program which scored best, we repeat the procedure, which results in a beam search with beam width 1. The search ultimately stops if $score_{P^k_p} < score_{P^k_{p-1}} \forall i$. As this may take long, we define a maximal depth $k_{max}$, meaning that we apply at most $k_{max}$ rules subsequently to a program, after which we stop repeating the search.

The procedure is illustrated in Figure 5.9. Each level corresponds to the application of one rule. The different $P^k_i$ correspond to the programs resulting from applying a rule to a program $P^{k-1}_i$. 

Figure 5.8: Example for the application of the Modify Comparison Rule to a program.
Figure 5.9: Potential search tree for applying rules. The arrows denote that one rule is applied, the resulting program with the best score is marked yellow and the final best ranked program is marked orange.

There will be a maximum of $k_{\text{max}} + 1$ levels, with level 0 being the original program. The final program is marked in orange. It is the program with the highest score overall.
5. Repair Rules
Implementation

This chapter discusses the implementation of our tool. Figure 6.1 illustrates the architecture of it. It consists of three main components: the abstraction, the interaction with the language model and the repair rules. A program \( P \) is given as input and the resulting output will be a series of changes one can apply to improve the program.

6.1. Overall Set-up

The tool is implemented in Java with the help of a few shell scripts to interact with the language model binaries. As every input program is processed the same way, the tool is embarrassingly parallel. Our data consists of programs written in Python 2.7, thus we operate on the Python

![Figure 6.1.: General overview over the architecture of our tool.](image-url)
6. Implementation

AST when computing the abstraction or applying rules. To parse the programs we use the parser from the Jython library v2.7b1.

When configuring the tool, one can choose between the two abstractions and between five modes:

- **Development.** The tool writes any output to stdout.

- **Training.** Compute the sentences for the input program(s) and write all sentences into a result file. Note that there will be only one output file generated, so it can be supplied as training data to ngram. The file will only contain sentences, there will be no information as to which object from which function from which program they belong to. This only uses the abstraction.

- **Test.** Compute the score(s) for the input program(s) using the language model trained earlier. The output will be written into a result file. This step does not yet apply the rules.

- **Modify.** Compute the score for the input program(s) using the language model trained earlier, then try to find a series of modifications to apply to improve the score. This is the normal operation for the tool. The output is written to a HTML file.

- **Print Detail.** This is useful for debugging. For a set of input programs, it will print a HTML file stating the program, its sentences and the resulting scores.

Note that Test and Modify will use the same abstraction as was used to compute the language model. You cannot mix language models when switching from Training to Test to Modify.

6.2. Abstraction

We implemented both of the abstractions discussed in Chapter 3 as intraprocedural static analysis, operating on the AST of each method of the programs. We assume all programs to be syntactically correct, as otherwise we are unable to obtain an AST for them. Additionally, our analysis is oblivious to certain semantic errors, such as undeclared variables or type errors, which are not reflected in the AST itself. Any information that is not in the AST is certain to be missing from our analysis. We use the Python AST, documentation on it can be found under [Python Software Foundation ].

A Python specific limitation is that Python does not have a `new` statement to indicate a new object is created. Thus, we have to treat every single function call as potentially creating a new object.

For now, when implementing the abstraction semantics, we ignore the more complex Python structures such as try-catch, yield, lambda functions, etc., as they are more complicated to analyse. Also, our test data does not contain these structures.

Let us look at the two abstractions separately in detail now.
6.2. Abstraction

6.2.1. Per-Object Semantics

The implementation is very straightforward. We use a points-to graph to keep track of aliasing and apply standard program analysis procedures for computing the abstraction. To this end, we use a Visitor Pattern to iterate over the AST. Since it is possible that the program has infinitely many traces, we apply loop bounding to reduce the number of traces. Also, we limit the number of nested branches we take. In our case, a loop unrolling factor of 2 and a nesting limit of 5 have worked well.

If desired, one can additionally bound the number of sentences kept, by randomly evicting some sentences when merging two set of sentences at a join point. However, we found that when comparing a incorrect program with its corrected version, using random sentences bounding conceals the relationship between a program and its fix. The abstraction does not reproduce the same set of sentences in every execution, because the sentences kept are chosen randomly. This means that in one execution, the incorrect program might score better than the corrected version, and in the next, it is the other way round. As we will need a reproducible score for each execution when searching for a better program by applying rules, we decided to avoid random sentences bounding.

Nonetheless, we need sentences bounding to speed up the computation. Therefore, we apply a simple type of bounding, where at join points, we do not merge additional sentences in if we already have more sentences than a certain threshold. We used a limit of 30 sentences to keep per object.

6.2.2. Dataflow Semantics

To facilitate control flow handling and to ensure the analysis is not mixing execution paths in the depends-on relation, we enumerate all traces and then execute a visitor on the AST nodes of this trace. In that way, we factor control flow out and focus on finding all dependency relations in the trace. However, we do not support Python’s ternary conditional operator, a so called ifExp, which returns an expression based on the condition as follows:

\[ r = \text{expression1 if condition else expression2} \]

This ternary operator cannot be easily discovered in the AST and thus would complicate the trace enumeration by far, so for now we do not process ifExp nodes of the AST.

We again bound the number of traces by applying loop unrolling and we limit the number of traces we enumerate. We use a loop unrolling factor of 2 and stop enumerating traces after 30 traces. Additionally, when constructing sentences, we bound the sentence length to 30 words.

The depends-on relation is computed lazily while extracting the sentences. To this end, we keep a map \( \text{varId} \rightarrow \mathcal{P}(\text{varId}) \), which maps each variable identifier to the variable identifiers it transitively depends on. The map gets updated after each statement evaluation, as described in Algorithm 6.1. Note that if the algorithm encounters an identifier which is not contained in the map, we simply initialize it to depend on itself: \( \text{map}_{\text{depends-on}}(\text{unk}) = \text{unk} \).

There is one limitation to this approach. We assume the left hand side of Assign, AugAssign, and For to be a simple terminal. But it could also be a subscript access \( l[i] \) or a tuple \((x, y)\).
Algorithm 6.1 Algorithm to lazily compute the depends-on relation while evaluating statements.

1: function EVALUATE_STMT(stmt)
2:   //map depends—on maps each object to all objects it transitively depends on and all objects
3:   // that transitively depend on it
4:   if stmt is Assign then
5:      // syntax: target = expr
6:      lhs ← EVALUATE_EXPR(stmt.target)
7:      rhs ← EVALUATE_EXPR(stmt.expr)
8:      map depends—on(lhs) ← rhs
9:   else if stmt is AugAssign then
10:      // syntax: target op= expr
11:      lhs ← EVALUATE_EXPR(stmt.target)
12:      rhs ← EVALUATE_EXPR(stmt.rhs)
13:      // register event target op expr
14:      map depends—on(lhs) ← lhs ∪ rhs
15:   else if stmt is For then
16:      // syntax: for target in iter: body; else: orelse
17:      target ← EVALUATE_EXPR(stmt.target)
18:      iter ← EVALUATE_EXPR(stmt.iter)
19:      // register event target In iter
20:      map depends—on(target) ← iter
21:   else
22:      // process stmt normally, but it will not update the map.
23:   end if
24: end function

In the latter case, we resolve this by simply treating both identifiers x and y to depend on every
object on the right hand side. In the former case, however, we do not have an identifier at hand
anymore. To this end, we create a new label ArrayAccessk and use it as key for the map instead
of l[i]. Unfortunately, in the following program

```python
def m(l, i):
    l[i] = 1
    x = l[i]
    return x
```

at the second line, we will miss the fact that l[i] depends on 1. We could overcome this by
applying a simple form of alias analysis and just compare if the subscript access is to the same
element by comparing the names of the collection (l) and the index (i), but this would only fix a
few cases. For this analysis to be useful, we would need to discover if the index or the collection
have been modified between the write and the read, and we still would miss dependencies such
as in

```python
def m(l, i):
    l[i] = 1
    i -= 1
    x = l[i+1]  # same element as in line 1.
    return x
```
To this end, we did not implement this simple form of alias analysis. This means that our computed depends-on relation is an under-approximation of the real depends-on relation.

6.3. Scoring

For building the language model and interacting with it, we use the SRILM toolkit [Stolcke 2002]. It comes as a collection of C++ libraries and binaries, as well as a set of command line utilities built on top of these libraries. SRILM constructs language models based on \( n \)-gram models and provides a wide range of parameters to tune the model.

We use the binary \texttt{ngram-count}, which is used to create the language model, and \texttt{ngram}, which estimates the probability of test data on a given language model.

We interface with them from Java with the help of a few wrapper scripts. To train the language model, we use the tool in the Training mode and then invoke the training script manually. We use the manual step because we want to have control over how our model is trained. Currently, we invoke \texttt{ngram-count} the following way:

```bash
ngram-count -kndiscount \ 
  -interpolate -text TRAININGDATA -order 3 \ 
  -write-vocab model.vocab -lm model.lm \ 
  -write-binary-lm
```

This means, we use modified Kneser-Ney discounting (\texttt{-kndiscount}) to cope with out of vocabulary words. Other options available are Witten-Bell (\texttt{-wbdiscount}), Good-Turing (\texttt{-gt}), or original Kneser-Ney (\texttt{-ukndiscount}). Our input is in \texttt{TRAININGDATA}, and we use a \( n \)-gram size of 3. We write out both the vocabulary and the language model, and we output the model in binary format. Binary models are faster to read and allow for even faster optimizations such as limiting the vocabulary used. Optionally, we also write out the \( n \)-grams, using the additional parameters

```bash
-writel model.counts1 \
```
6. Implementation

```
-write2 model.counts2 \\
-write3 model.counts3
```

To test the language model, we interface with ngram from Java. Because our tool is parallelized, we run the language model as a server. So far, the server start-up is done manually, it could however be easily integrated into the tool. The server is started using the command

```
ngram -order 3 -vocab model.vocab \\
   -lm model.lm -server-port 8004
```

The language model can then be queried by sending sentences to port 8004. Our query script, which we invoke from within our tool, looks the following:

```
ngram -ppl TESTDATA -order 3 -vocab VOCAB \\
   -escape "#" -use-server 8004 \\
   -cache-served-ngrams -debug 1
```

The parameter VOCAB specifies the language model’s vocab depending on the abstraction used, and the parameter -escape indicates which character is used to indicate comments in the TESTDATA file. We use comments to indicate which sentences belong to which program, which is useful for debugging. The parameter -debug indicates how much output we expect from our language model. -debug 1 prints the sentence along with its probability, perplexity, and some other information. A higher value would show the detailed probabilities of the n-grams for the sentence. As we do not need so much detail, we use value 1.

We capture the output of the query script in Java and do post-processing such as normalization of the logProb and computing the average there. This makes interfacing with the rules and the search procedure easier.

For more details on SRILM, refer to the manual pages.[Stolcke]

### 6.4. Repair Rules

We implemented all of the rules described in Section 5.2, but due to the time constraint on the thesis, we had to hard-code them. This means, that they are currently only available in Java source code and operate directly on a subtree of the AST of the function we are currently analysing. The architecture is simple, we designed an abstract base class Rule, from which each rule inherits. The rules then are applied to the AST of the original program, as described in Algorithm 6.3. Because the rules operate on a subtree of the AST of the function under analysis, it is essential to produce a copy of the whole AST beforehand, such that each rule only modifies "its own" copy. Otherwise we would apply multiple rules at once, which is not what we want.

However, this set-up is not optimal. In order to add new rules, one needs to hard-code them by extending the base class Rule. It is desirable to have some sort of description language, through which one can easily add new rules or modify the existing ones without knowledge of the implementation. This is left as future work.
Algorithm 6.3 Algorithm to apply rules to AST.

**Input:** orig: AST of original program

**Output:** Set of modified ASTs, each resulting from applying one rule to orig.

1: results ← new set
2: for each Rule rule do
3:   copy ← deep copy of orig
4:   rule.applyTo(copy)
5:   add copy to results
6: end for
7: return results

Also, some rules such as the Insert Test Rule and the Remove Functions Rule require some configuration in order to be applied. For the Remove Functions Rule, we just state the name of the undesired function and the argument index with which it should be replaced, e.g. "round, 1", which will remove calls to round and replace them with the first argument. This works great, and conforms to the duck typing in Python’s type system. For another language, it might be desirable to state the qualified function name to ensure only the desired API is removed.

The Insert Test Rule however needs a more elaborate configuration possibility, and would therefore profit from a description language. In order to handle whole expressions which have to be guarded by a test, we need to be able to specify placeholders for variable names. A simple example would be a subscript access s[0], where we need to ensure that s is not empty. But s is a variable identifier, so when configuring the rule for such accesses, we would need to be able to state something like "$1[0] \rightarrow \text{len}($1) > 0". This is another application of a potential description language, which as mentioned before, is left as future work.
6. Implementation
Experiments and Evaluation

In this chapter, we will discuss the experiments we ran on our tool and their outcome. We will explain the settings for the experiments, state our hypotheses, and discuss the results. We will provide examples where the prediction works well and show up cases where it does not.

7.1. Set-up

We train our language model on 1 GB of data which we downloaded from GitHub. We assume that these Python projects consist of correct programs. Building the file which contains all sentences from these 1 GB of programs takes about 10 minutes of time and ngram requires again few minutes. Overall, the language model can be built in roughly 15 minutes, which is acceptably fast in our opinion.

Figure 7.1 depicts some statistics for our training data. They show that most sentences have between 1 and 5 words. As we cut off the sentences after 31 words, we receive comparatively many sentences with 31 words, in contrast to e.g. between 20 and 25 words. The per object abstraction has about 7 times as many sentences as the dataflow abstraction, which can again be attributed to the amount of new objects created in this abstraction.

As test data, we have 54 528 submissions from the MIT open online course "Introduction to Computer Science and Programming Using Python". These programs can be categorized into eleven tasks. For each task, the submissions include both incorrect and correct programs. The task oddTuples has 32 837 submissions, the two tasks applyToEach and biggest have one submission each, and the remaining tasks have between 1113 and 4537 submissions. However, not all of the tasks were suitable to our approach. The two tasks simpleHangman and hangman required interaction with the user such as querying for input, which we cannot provide.
7. Experiments and Evaluation

Figure 7.1.: Overview over the distribution of sentence length for both abstractions.

(a) Distribution of the 5,091,508 sentences obtained through dataflow abstraction

(b) Distribution of the 34,543,164 sentences obtained through per object abstraction

Figure 7.2.: Comparison of average score for correct programs (orange) and incorrect programs (blue) to the task computeDeriv.

We remain with 7 tasks which suit our approach: oddTuples, evaluatePoly, getAvailableLetters, getGuessedWord, isWordGuessed, computeRoot, and computeDeriv. The task description for each of these tasks can be found in the Appendix.

7.2. The Separation Experiment

As mentioned in Chapter 4 and Section 5.1, our first idea was to find an abstraction which can separate the submissions into correct and incorrect programs. From this separation, we wanted to draw information about potential rules. In order to see if we can achieve separation, we did the separation experiment.

First, we ran our tool with per object semantics on all submissions to a task, both correct and incorrect ones. We compared the average of incorrect programs to the average of correct programs, but instead of showing signs of separability, they shared the distribution. This is illustrated in Figure 7.2.

For the reasons stated in Section 4.2, the separation did not work out. The results when using the dataflow abstraction look identical. Therefore, we abandoned the idea of separability.
7.3. The User Study

In Section 4.2, we present a new hypothesis: For incorrect programs, the corrected version will achieve a higher score. To evaluate if our abstraction satisfies this hypothesis, we conducted a user study. We took the three tasks `evaluatePoly`, `getAvailableLetters`, and `oddTuples` and picked 100 incorrect programs from each at random. We then presented these 300 programs to a group of programmers and asked them to fix them while trying to maintain the coding style. These fixed programs were then evaluated under the abstraction and compared to their incorrect counterpart.

The programmer group consisted of 17 programmers, thereof three ETH master students, one ETH alumnus, six ETH PhD students and one ETH professor. Furthermore, five PhD students from the Technion in Israel have participated in this study. One programmer remains anonymous.

Table 7.1 and Table 7.2 show the results of the study. Programs categorized as `willnotfix` were either empty submissions, completely off the task, or require too many changes to be made to be worth the try. Because we cannot fix them, we will ignore them in the evaluation. The percentages denoted with % are relative to the total without the `willnotfix` programs, given in brackets in the column `total`.

The first overview over the results is encouraging. We were able to improve the score in over two thirds of the programs. The two abstractions each perform well, however, they score the programs differently. There are programs which rank better under the dataflow style abstraction, but worse under the per object abstraction, and vice versa.

Note that the results for the per object semantics have to be taken with a grain of salt. Because
7. Experiments and Evaluation

import math
def evaluatePoly(poly, x):
    sum = 0.0
    for i in range(len(poly)):
        sum += (poly[i] * 1.0) * ((x * 1.0) ** i)
    return sum

(a) Original Program with a score of $-1.2947596$.  
(b) Programmer’s fix with a score of $-1.3345623$.  
(c) Fix which maintains code style, with a score of $-1.2844366$.

Figure 7.3: Example of a fix under dataflow style abstraction which did not maintain the coding style of the student.

computing the abstraction with all sentences is not feasible within reasonable time, we had to implement sentence bounding. At join point, if we have more sentences than a threshold, we will not merge additional sentences into our set of sentences. This non-random way was chosen to guarantee reproducibility for the comparison.

For the predictions that did not score well, we can distinguish between three cases: wrong fix for the program, lower score of the correct version, and inability to distinguish the program and its fix.

A few of the programs scored worse, because the programmer did not maintain the coding style. An example under the dataflow abstraction can be seen in Figure 7.3. We see that the programmer removed a multiplication with 1.0 twice, which, while not bearing any benefit, does not influence the computed result. However, the sentences generated from the original program will contain Mult events related to this, which cause the sentences to have a higher probability. This originates from our training data, containing several numerical projects that themselves contain many mathematical operations in their sentences. 7 programs scored worse for this cause.

In the second case, the tool actually assigned the correct program a lower score. Manual inspection of the results shows that there are various causes for this behaviour. Some programs contained in the fixed version more traces because control flow was added. Figure 7.4 displays an example of this case, under the dataflow abstraction. Due to the wrongly indented return statement, the original program has only a subset of the traces of the fixed version, and the additional traces are ranked slightly below the original score, so the overall fix will receive a lower score than the original.

Fixes to the task oddTuples involve changing the variable type to Tuple. In the sentences, this introduced a TUPLE_ELEMENT word which appears to drag the overall score down.

In general, there is no single common reason for these cases.
7.3. The User Study

```python
def evaluatePoly(poly, x):
evaluatedTerm = 0
total = 0
for index in range(0, len(poly)):
evaluatedTerm = (poly[index] * (x ** index))
index += 1
total = total + evaluatedTerm
return total
```

(a) Original program, with a score of $-1.351407$.

```python
def evaluatePoly(poly, x):
evaluatedTerm = 0
total = 0
for index in range(0, len(poly)):
evaluatedTerm = (poly[index] * (x ** index))
index += 1
total = total + evaluatedTerm
return total
```

(b) Fix which results in more traces, with a score of $-1.433836$.

**Figure 7.4.** Example of a fix under dataflow style abstraction which introduces more sentences, which lowers the score.

```python
def oddTuples(aTup):
return aTup[0: -1: 1]
```

(a) Original program.

```python
def oddTuples(aTup):
return aTup[0::2]
```

(b) Fix which results the same sentences.

**Figure 7.5.** Example of a fix under per object abstraction which does not reflect as change in the sentences, and thus leads to the identical score.

Lastly, the abstraction might not be able to distinguish between the correct and the incorrect program. This is what happens for the task `getAvailableLetters`. Most of the programs that received the same score were fixed by adding an import statement. However, our analysis is intraprocedural and hence does not capture if there was an import somewhere. The sentences generated for the programs will be identical, and for this reason, the score remains the same.

Another case where the sentences remain identical is when encountering an uninitialized variable under the dataflow style semantics. Adding an initialization statement of the shape `var = constant` will not touch the sentences extracted from the program.

For per object semantics, exchanging one constant for another can lead to the sentences remaining identical. Figure 7.5 illustrates an example for this. Because the constants are only used at that line of code, their sentences do not change.

Overall, we are satisfied with the results from this experiments.
7. Experiments and Evaluation

<table>
<thead>
<tr>
<th>Task</th>
<th>TP</th>
<th>FP</th>
<th>FN</th>
<th>TN</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>oddTuples</td>
<td>45</td>
<td>43</td>
<td>6</td>
<td>37</td>
<td>0.51</td>
<td>0.88</td>
<td>0.63</td>
</tr>
<tr>
<td>evaluatePoly</td>
<td>50</td>
<td>39</td>
<td>1</td>
<td>49</td>
<td>0.57</td>
<td>0.98</td>
<td>0.71</td>
</tr>
<tr>
<td>getAvailableLetters</td>
<td>17</td>
<td>56</td>
<td>0</td>
<td>50</td>
<td>0.23</td>
<td>1</td>
<td>0.54</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>112</td>
<td>138</td>
<td>7</td>
<td>136</td>
<td>0.45</td>
<td>0.94</td>
<td>0.63</td>
</tr>
</tbody>
</table>

*Table 7.3.: Results from running the tool in MODIFY mode under the dataflow abstraction.*

7.4. The Synthesis Experiment

After establishing that the fixed version of a program has the better score, we want to see if this works as an automated process as well. To this end, we define a new hypothesis: Good programs are more resilient to changes than bad programs. This means that we are more likely to successfully apply a rule to a bad program than to a good program. To discuss this hypothesis, we run our tool on the 300 incorrect programs from the user study and manually inspect if the reported results support the hypothesis.

To categorize the findings, we consider each location a rule should have been applied. If the rule was applied properly, we call it a true positive (TP). Applying a rule where it is not necessary is therefore a false positive (FP). It means that the program’s score improved even though the rule worsened the program’s correctness. If the tool did not apply a rule where it should have, it is a false negative (FN), stating that the score decreased although the correctness increased. And if there was no rule to be applied, we call it a true negative (TN). Because our tool is able to suggest multiple rule applications for one program, we have in total 393 locations in 300 programs to consider.

We run dataflow abstraction with a search depth $k_{max}$ of 4.

The results can be found in Table 7.3. We achieve an overall precision of 0.45 and a recall of 0.94, with an accuracy of 0.63. Similar to the user study, our tool works correctly in two thirds of the cases. The recall tells us that for almost every location where we should apply a rule, we do so. This is good because it shows us that for bad programs, the rules will be applied. However, the rather low recall indicates that the hypothesis does not entirely hold. Manual inspection showed that this is mostly caused by the invert condition rule and the loop iterate rule.

Table 7.4 gives an overview over which rule was applied how many times correctly and incorrectly in each task. We use the following abbreviations: $Ret =$ Return Rule; $LI =$ Loop Iterate Rule; $IC =$ Invert Condition rule; $MC =$ Modify Comparison Rule; $RF =$ Remove Functions Rule; $IT =$ Insert Test Rule. A plus (+) indicates that the rule was used at the correct location, and a minus (−) signifies the usage of the rule where it should not be used.

We notice two things. First, the Loop Iterate Rule has a high number of times where it should not be used. For the task oddTuples, the Loop Iterate Rule works acceptable, but for the other tasks, it is not necessary to be applied. This can be attributed to the fact that only the task oddTuples requires to iterate over the indices of a collection, as the Loop Iterate Rule establishes. Therefore, we now believe that the Loop Iterate Rule is more of a task-specific rule than a general correctness rule.
Similarly, we notice that the Invert Condition Rule, while working in a few cases, is not helping in many cases. It does not even show a tendency towards a task or indicate that a certain condition, e.g. ≠, is always preferred over another. This leads us to believe that this rule is not too helpful as a general correctness rule.

Overall, we can say that our tool works surprisingly well, especially given that it has only a limited amount of fixes it can apply and that it possesses no knowledge about the tasks its input programs have to solve.

Table 7.4: Number of applications for each rule.
7. Experiments and Evaluation
Conclusion and Future Work

We presented a new approach at automatically creating individual feedback for submissions of massive open online courses. We combine program analysis with statistical language modelling to create a model representing correct programs. The model is trained on a large number of open source projects. We then use the model to score submissions. Our results indicate that in more than two thirds of the cases, for an incorrect program, a correct version of this program scores better.

Furthermore, we provide a synthesizer which makes use of the model to find fixes for incorrect programs which increase the score of the program. Our tool uses rules drawn from common bugs and work with precision 0.45, recall 0.94 and an accuracy of 0.63.

There are multiple directions in which the work from this thesis can be extended. The first direction is to refine the abstraction. To this end, one can implement the knobs discussed in Section 3.4.

Considering the scores, we propose to think of a different metric to combine the scores of individual sentences to a score for whole programs. Additionally, one can experiment with different smoothing methods and $n$-gram orders.

A very important step for program synthesis is to devise a description language for the rules and their configuration. Using this, more rules and more instantiations of the existing rules can be applied.

Further experiments should include another evaluation of the effectiveness of the rules (from the perspective of the Top 5 ranked programs which are created while applying modifications). We would also like to check if for a given program, its corrected version ends up being listed in the top 5 most likely suggestions (and similarly for a correct initial program).
8. Conclusion and Future Work
Appendix

This is a short overview over the tasks contained in our test data. They belong to the edx course "MITx 6.00.1x Introduction to Computer Science and Programming Using Python". We will give a short specification on each of the tasks we have used.

A.1. computeDeriv

computeDeriv(poly) requires as input a nonempty list of coefficients to a polynom. It will return the derivative of the polynom poly as a list of floats, or [0.0], if the derivative is 0. For example:

```python
>>> poly = [-13.39, 0.0, 17.5, 3.0, 1.0]
    # - 13.39 + 17.5x^2 + 3x^3 + x^4
>>> print computeDeriv(poly)
[0.0, 35.0, 9.0, 4.0]
    # 35^x + 9x^2 + 4x^3
```

A.2. computeRoot

computeRoot(poly, x_0, epsilon) computes the root of a function according to Newton’s method. It expects a list of coefficients to the polynom poly, a starting point $x_0$ and a error threshold epsilon, and will return a list of type [float, int]. The float denotes the root, and the int states the number of iterations it took to compute the root.
A. Appendix

For example:

```python
>>> poly = [0,0,1] #x^2
>>> x_0 = 2
>>> epsilon = 0.1
>>> print computeRoot(poly, x_0, epsilon)
[0.25, 3]
```

A.3. evaluatePoly

`evaluatePoly(poly, x)` will evaluate the polynom described by the coefficients in `poly` at `x` and return the result as a float.
For example:

```python
>>> poly = [1,0,1] # 1 + x^2
>>> x = 2
>>> print evaluatePoly(poly, x)
5.0
```

A.4. getAvailableLetters

The input to `getAvailableLetters(lettersGuessed)` is a list of letters which have been guessed so far. The output should be a string consisting of all letters of the English alphabet, which have not yet been guessed.
For example:

```python
>>> lettersGuessed = ["b","a","n","a","n","a"]
>>> print getAvailableLetters(lettersGuessed)
'cdefghijklmopqrstuvwxyz'
```

A.5. getGuessedWord

`getGuessedWord(secretWord, lettersGuessed)` takes as input a string `secretWord` and a list of letters which have been guessed so far. It will return a string consisting of the secret word, where each letter which has not yet been guessed is replaced with "_".
For example:

```python
>>> secretWord = "bapple"
>>> lettersGuessed = ["a", "b", "c", "d", "e"]
>>> print getGuessedWord(secretWord, lettersGuessed)
'ba___e'
```
A.6. isWordGuessed

isWordGuessed(secretWord, lettersGuessed) requires the same input as getGuessedWord, but will instead return True, if all letters in secretWord have been guessed, or False otherwise.

For example:

```python
>>> secretWord = "potato"
>>> lettersGuessed = ["o","r","a","n","g","e"]
>>> print isWordGuessed(secretWord, lettersGuessed)
False
```

A.7. oddTuples

oddTuples(aTup) takes as input a tuple aTup and returns another tuple consisting of every other element of aTup.

For example:

```python
>>> aTup = (4,3,2,1,5)
>>> print oddTuples(aTup)
(4,2,5)
```
A. Appendix
Bibliography


Eigenständigkeitserklärung


Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

**Titel der Arbeit** (in Druckschrift):

**AUTOMATED TUTORING FOR MASSIVE OPEN ONLINE COURSES**

**Verfasst von** (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller VerfasserInnen und Verfasser erforderlich.

**Name(n):**

**ZELLER**

**Vorname(n):**

**CHRISTINE**

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt "Zitter-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

**Ort, Datum**

**Zürich, 17.9.14**

**Unterschrift(en)**

**C. Zeller**

Beim Gruppenarbeiten sind die Namen aller VerfasserInnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.