FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Automated Feedback Generation in Introductory Programming Education

A Dynamic Program Analysis Approach

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Ivan Radiček, BSc. MSc.
Registration Number 1329038

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Math. Dr.techn. Florian Zuleger

The dissertation has been reviewed by:

Johannes Kinder

Tao Xie

Vienna, 6th February, 2020

Ivan Radiček

Erklärung zur Verfassung der Arbeit

Ivan Radiček, BSc. MSc.
Dresdner Straße 117
1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. Februar 2020

Ivan Radiček

Acknowledgements

This thesis, and my whole PhD journey, would not be possible without support of numerous people.

First of all, I would like to thank my advisor Florian Zuleger. Florian believed in my abilities and accepted me to the PhD position when I had no experience in research or formal methods. For the whole duration of my studies he showed great patience with our paper submissions and writing of this thesis, while helping me to express my ideas clearly and teaching me the importance of practical examples. Finally, I want to thank Florian for his general friendliness and readiness to help with various issues, like finding ways to extend my contract at the university.

I would also like to thank my unofficial co-advisor Sumit Gulwani. Sumit has introduced me to the topic of introductory programming education. Our countless phone discussions, and Sumit's enthusiasm, encouragement and guidance have tremendously contributed to my research.

Next, I would like to thank all the colleagues and members of the Forsyte group, LogicCS program, and all the people from the 3rd floor of Favoritenstraße 9-11. The conferences and summer schools that we have jointly attended, discussions, lunches, dinners, board game nights, beers . . . all of these played a significant role during my time at the university. I would also like to thank the office and system administrators of the Forsyte group for their valuable support with the paperwork and technical issues.

Finally, I would like to thank my family, girlfriend and friends for their great moral and emotional support and enormous patience over these past several years.

Kurzfassung

Wer jemals Übungsaufgaben korrigiert oder bewertet hat, ist sich darüber bewusst wie *mühsam, fehleranfällig und zeitaufwändig* diese Aufgabe ist. In Bezug auf *Programmieraufgaben* ist die Situation noch gravierender: aufgrund des steigenden Bedarfs an ProgrammiererInnen gibt es Online-Kurse mit Tausenden von Studenten, was manuelles Bewerten schlicht unmöglich macht. Demgegenüber haben gerade Studenten, welche *noch nie zuvor programmiert haben*, einen hohen Bedarf an Betreuung. Diese Ausgangslage hat zu einer Vielzahl an Forschungsansätzen geführt wie man computergestützt Feedback für Programmieranfänger generieren kann. Viele Ansätze basieren auf der Praxis und den Methoden des Software Engineering, des Software Testens und der Programmverifikation: Code Reviews, Debugging, automatische Testfallgenerierung, Programmsynthese, Programmreparatur, . . . Dieses Feedback ist sehr nützlich da es nachahmt wie ProgrammiererInnen mit ihrem Code arbeiten und darüber nachdenken.

Dennoch verbleiben eine Vielzahl von Herausforderungen. In dieser Arbeit addressieren wir die beiden folgenden Fragestellungen:

- Die bisherige Forschung hat vor allem die Generierung von funktionalem Feedback behandelt, das heißt Studenten dabei zu helfen *ein korrektes Programm* zu schreiben, wobei nicht-funktionale Eigenschaften wie zum Beispiel *Performance* ignoriert werden.
- Existierenden Ansätzen fehlt mindestens eine der folgenden Eigenschaften: *Automatisierung* (so wenig manueller Aufwand wie möglich), *Korrektheit* (die Generierung von korrektem Feedback), *Performance* (die Generierung von Feedback innerhalb weniger Sekunden), *Vollständigkeit* (die Generierung von Feedback für die meisten studentische Lösungen), und *Nutzen* (Reduktion des Aufwands für den Betreuer oder Erzielen von Fortschritt durch die Studenten).

Wir studieren zunächst ineffiziente (aber korrekte) studentische Lösungen um zu verstehen welche Performanceprobleme bei Programmieranfängern auftreten. In dieser Studie stellen wir fest, dass es zur Feedbackgenerierung erforderlich ist, *die algorithmische Strategie* einer Lösung zu erkennen, aber Implementierungsdetails zu ignorieren. Zu diesem Zweck entwickeln wir eine leichtgewichtige Spezifizierungssprache, welche es dem Lehrer erlaubt algorithmische Strategien zu spezifizieren, und eine neue *dynamische Programmanalyse*, welche es ermöglicht zu überprüfen ob ein studentischer Lösungsansatz eine Spezifikation des Lehrers erfüllt.

Um Feedback für die funktionalen Korrektheit von studentischen Lösungen zu generieren, entwickeln wir einen neuartigen voll-automatiserten Reparaturalgorithmus, welcher einem *Wisdom of the Crowd* Ansatz folgt: unser Algorithmus nutzt bereits existierende korrekte studentische Lösungen um syntaktische Änderungen an inkorrekten studentischen Lösungen zu generieren. Der Reparaturalgorithmus basiert auf und erweitert unsere dynamische Programmanlyse um zu überprüfen ob ein studentischer Lösungsansatz das selbe Verhalten hat wie eine bereits existierende korrekte Lösung.

Wir haben beide Ansätze in praktischen Tools implementiert und an einer großen Anzahl von studentischen Lösungen aus verschiedenen Programmierkursen ausgwertet. Wir evaluieren beide Tools im Hinblick auf die zuvor beschriebenen wünschenswerten Eigenschaften: Automatisierung, Korrektheit, Performance, Vollständigkeit und Nutzen.

Abstract

Anyone who ever manually corrected or graded any type of student assignment is aware of how *tedious, error-prone and time-consuming* this task is. In the context of *programming* assignments the situation is even worse: with a rising demand for programming education there are online courses with thousands of students, which makes manual grading downright impossible. However, students who are learning programming, especially at the *introductory* level, are in pressing need of guidance. This has motivated a lot of research on computer-aided feedback generation in introductory programming education. Many of these approaches are based on practices and methods employed in software engineering, testing, and verification, such as: code reviews, debugging, automated test generation, program synthesis, program repair, . . . This is very useful feedback as it mimics how software engineers reason about the code.

However, there are still numerous open issues. In this thesis, we will address the following two challenges:

- Most of the existing research focused on generating functional feedback, that is, to help the students write *any correct program*, while ignoring non-functional properties such as e.g., *performance*.
- Most of the existing approaches lack some of the following desirable properties: *automation* (require as little manual effort as possible), *correctness* (generate correct feedback), *performance* (generate feedback in order of seconds), *exhaustiveness* (generate feedback for most student programs), and *usefulness* (reduce the teacher effort or help the student to make progress).

We first study inefficient (but correct) student programs in order to understand the performance problems faced by students in introductory programming. From the study we observe that in order to provide feedback we need to identify the *high-level algorithmic strategy* used by the student's program, while ignoring its low-level implementation details. To that end we propose a lightweight specification language that enables the teacher to specify various algorithmic strategies, and a novel *dynamic program analysis* to determine whether a student's attempt matches the teacher's specification.

To provide feedback on (functional) correctness of student programs we propose a novel fully-automated program repair algorithm that uses the *wisdom of the crowd*: it leverages

the existing correct student solutions to syntactically modify incorrect student attempts. The repair algorithm builds on and extends our earlier dynamic program analysis in order to determine whether a modified (repaired) program has the same behavior as a correct solution.

We have implemented both of the proposed approaches in practical tools and performed an experimental evaluation on a large number of student attempts from various programming courses. We have evaluated both of our approaches on the above mentioned desirable properties: automation, correctness, performance, exhaustiveness, and usefulness.

Contents

CHAPTER 1

Introduction

Providing feedback on and grading of student assignments in science, technology, engineering and mathematics (STEM) fields is error-prone and time-consuming task for a human teacher. This problem presents itself even in a classroom setting, but it is exacerbated by the rising popularity of *Massive Open Online Courses* (MOOCs) [Mas11].

However, providing feedback is an integral part of any class; in addition to helping students understand the subject material better, immediate feedback can also enable new pedagogical benefits [Gul14], such as:

1. Allowing immediate re-submission opportunity to students who submit imperfect solutions.

2. Providing immediate diagnosis on class performance to a teacher who can then adapt her instructions accordingly.

This, along with rapid advances in computer science, has motivated a lot of research on providing computer-aided feedback. These research efforts can be classified along several dimensions:

- *Subject of assignments*, such as introductory programming [SGSL13; Til+13a; Kim+16], automata theory [Alu+13], mathematical procedural problems [AGP14], proofs [AGK13] and geometry [GKT11].

- *Level of feedback automation*, that is, whether the feedback is automated, semi-automated or manual.

- *Aspects on which the feedback is provided*, such as functional correctness, and non-functional properties (e.g., performance characteristics).

- *Nature of the feedback*, such as counterexamples, error localization, repair suggestions, or manual human feedback.
- *An underlying technology*, such as static or dynamic program analysis, program synthesis, or probabilistic methods.

In this thesis we focus on the problem of automated and semi-automated feedback generation for introductory programming assignments. We also focus on using dynamic program analysis [1].

1.1 Feedback Generation for Introductory Programming

We give a brief overview of this field, to better understand current research and its challenges.

A report from 2014 [2], mentioned in a recent paper about providing feedback on introductory programming [Kim+16], predicts that there will be one million more computer programming jobs than computer science students by 2020. This results in a huge demand for computer science education, which universities are, however, unable to meet.

In turn, this has resulted in many online platforms that provide computer science education, such as *edX* [3] and *Coursera* [4]. However, in contrast to traditional university classrooms, these online classrooms can have thousands of students, and therefore students cannot anymore expect personal feedback on their assignments by the human teaching personnel.

Hence, there is a huge need to automate providing feedback, class diagnosis and grading in introductory programming education. If full automation is not possible, semi-automated tools that help the teachers to deal with a large number of students, are also desirable.

Setting Next, we briefly describe the usual setting in which introductory programming classes operate to further understand the problem. This is, for example, the setting used in an introductory programming class *ESC 101 at IIT Kanpur, India*, where they use the *Prutor* [Das+16] web platform.

Students are assigned a programming problem, with a textual specification describing the problem. The specification describes *functional* requirements, that is, *what* a solution to the problem should do. The specification can also prescribe certain *non-functional*

[1] *Dynamic* program analysis reasons about a program by examining concrete executions of the program, opposed to *static* program analysis that examines program text (or to be more precise, its abstractions).

[2] "Analysis: The exploding demand for computer science education, and why America needs to keep up" (`http://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/`)

[3] `https://www.edx.org/`

[4] `https://www.coursera.org/`

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length)
    return false;
  foreach (Char ch in s.ToCharArray()){
    if (countChars(s, ch)
           != countChars(t, ch)){
      return false;
    }
  }
  return true;
}
int countChars(String s, Char c){
  int number = 0;
  foreach (Char ch in s.ToCharArray()){
    if (ch == c){
      number++;
    }
  }
  return number;
}
```

(a) An attempt to the anagram problem (**C1**).

```
def computeDeriv(poly):
  new = []
  for i in xrange(1,len(poly)):
    new.append(
        float(i*poly[i]))
  if new==[]:
    return 0.0
  return new
```

(b) An attempt to the derivatives problem (**I1**).

Figure 1.1: Examples of student attempts.

requirements, that is, *how* a solution should be written. We illustrate on an example below functional and non-functional requirements. Students submit (for example, through an online interface) their *attempts* (implementations of the given specification). After submitting an attempt, they might receive feedback, either immediately, or after some time period. When the feedback has been provided, they might be allowed to submit a new attempt.

We give some examples of various requirements.

Example 1.1. *Examples of functional requirements are:*

(F1) Given two strings `s` *and* `t`*, determine whether they are anagrams. Two strings are anagrams when one can be permuted (rearranged) to match the other.*

(F2) Given a polynomial (encoded as a list of floating point coefficients), compute its derivative (also encoded as a list of floating point coefficients).

(F3) Given a number, determine whether the sum of cubes of its digits is equal to the number itself.

Examples of non-functional requirements are:

(N1) Algorithmic complexity of the solution should be in $\mathcal{O}(n)$*.*

(N2) The program should not use any external libraries (or only some libraries).

(N3) The program should use some good coding practices (e.g., good variable and function names, comments).

For the discussion below we fix the <u>anagram</u> problem specified by the functional requirement (F1) and the non-functional requirement (N1); and similarly the <u>derivatives</u> problem specified by the functional requirement (F2).

Next, we give some examples of student attempts on the above problems and discuss how these attempts satisfy the problems' requirements.

Example 1.2. *Figure 1.1 shows two attempts: the attempt **C1** in (a) is written (in* C#*) for the anagram problem, while the attempt **I1** in (b) is written (in* Python [5]*) for the derivatives problem.*

*The attempt **C1** satisfies the functional requirement (F1); however, it does not satisfy the non-functional requirement (N1), since its complexity is in $\mathcal{O}\left(n^2\right)$.*

*The attempt **I1** does not satisfy the functional requirement (F2) because it returns wrong results on inputs of length ≤ 1 - it returns the floating point `0.0`, instead of the list `[0.0]`.*

Finally, we give some examples of feedback and discuss it on the above student attempts.

Example 1.3. *A possible feedback for the attempt to the anagram problem might be:*

- *"Your attempt is in $\mathcal{O}\left(n^2\right)$, while it can be in $\mathcal{O}\left(n\right)$."; or*
- *"Calculate the number of characters in each string in a preprocessing phase, instead of in each iteration of the main loop."*

A possible feedback for the attempt to the derivatives problem might be:

- *"On the input `[3.3]`, your attempt produces the output `0.0`, while the expected output is `[0.0]`."; or*
- *"The expression `0.0` on the line 7 of your attempt is wrong."; or*
- *"The expression `0.0` on the line 7 of your attempt should be changed to `[0.0]`".*

These examples are an illustration of what a potential feedback might look like, that is, there are not taken directly from some existing feedback generating system.

[5]We point out that the two attempts are in two different languages (C# and Python). This is because the real students attempts are in these languages and our approaches work on these respective languages. Further, we use these attempts throughout the thesis as running examples.

1.1.1 Connection to Software Testing and Analysis

Providing feedback in this setting is quite similar to software testing and analysis in traditional software engineering. Software engineers, as well, write an implementation for a certain specification, and get feedback from testing and analysis tools. However, compared to the usual software testing and analysis, introductory programming setting comes with certain simplifications, but also with certain additional challenges [SGSL13; Dru+14]:

- In the introductory programming setting *the complete specification is already known in advance*, while in traditional software engineering there is, at best, only a test suite or a partial specification. Further, it is safe to assume that in the introductory setting the course instructor has provided *a reference solution*; however, unlike experienced programmers, students who learn programming, are more likely to *misunderstand the specification.*
- *Errors are predictable* in the introductory setting, since students are solving the same assignment, after they have, most likely, attended the same lecture. However, again, unlike experienced programmers, beginner students will make mistakes that are unlikely in real-world programs.
- *There are a lot of correct solutions* available in the introductory setting from the students themselves. This is especially the case in courses that are repeatedly offered each year.
- *Programs in introductory setting are small*, therefore one can use techniques that are unlikely to scale on larger programs.

Next, we discuss some of the existing approaches to providing feedback and grading in introductory programming assignments. A more detailed overview of the related work is given in Chapter 5.

1.1.2 State of the Art

Peer-feedback A standard procedure in software engineering, when an engineer submits a unit of work (usually a commit or pull request [6]) to be merged into the code base, is to perform a code-review [7]. The review is performed by one or more colleague engineers. The review usually consists of multiple rounds, where the reviewers provide comments (suggestions for improvement) and the engineer who wants to merge the code

[6]A usual workflow is as follows:

1. An engineer works on a local copy of the code.
2. When a small unit of work (usually a single new feature or a bug fix) is done, the engineer submits her changes to be merged into the main code base.

[7]See `https://en.wikipedia.org/wiki/Code_review`.

acts upon them by changing the code to be merged. These rounds continue until the reviewers accept or reject the code to be merged.

This approached has been mimicked in providing feedback, in form of *peer-review* or *peer-feedback* [SGSL13; KK12]. The idea is that students review and provide feedback on other students' attempts. This idea has been further developed: instead of asking students to review code of their peers, one can also use *crowdsourcing* [Wel+12] for providing feedback and grading.

However, these approaches have several drawbacks:

- Students often have to wait for a response for a long period of time.
- A second, and more severe, problem is that there are no quality guarantees on feedback. That is, feedback might be incomplete, or even wrong; this is especially the case when students, who are themselves yet learning the material, provide feedback.

Failing test-cases Another standard practice in software engineering is *(unit) testing*: a code unit (e.g., a function or a module) is tested to conform to some specification. For simplicity, assume that a unit of code is a function and a specification is a set of input-output pairs, that is, pairs of inputs and expected outputs for each input. [8] If a function under test, for some input, produces an output that is different than the expected output, this means that the functions fails to match the specification, and it needs to be corrected. The benefits of such an approach are that tests are easy to run, and it is easy to find errors. [9]

Such an approach is easy to mimic in feedback generation. For a given problem (for simplicity, assume that it asks to write a single function) the students are presented with a failing test case. The test cases can be generated automatically [Til+13a] or selected from a comprehensive collection of representative test cases provided by the course instructor. [10] This is useful feedback, especially since it mimics the setting of how software engineers debug their code. Also, currently this is the most common way of providing feedback to students in practice.

However, this is usually not sufficient, especially for students in an introductory programming class, who are looking for more guided feedback to make progress towards a correct solution.

[8]In practice this can be more complicated. For example: a testing environment needs to be initialized; instead of a single expected output, a number of assertions is written for each test; different, possibly non-functional, side-conditions could be tested as well.

[9]The problem of testing in software engineering is that it is non-exhaustive, that is, it is easy to miss or omit a test case that would reveal an error.

[10]Practical experience shows that it is not difficult to write a comprehensive set of test cases for problems in introductory education, as opposed to general software engineering. We further discuss this issue later in the thesis.

```
1 def computeDeriv(poly):
2   new = []
3   for i in xrange(1,len(poly)):
4     new.append(
5         float(i*poly[i]))
6   if new==[]:
7     return [0.0]
```

(a) An attempt to the derivatives problem with a missing statement.

```
1 def computeDeriv(poly):
2   new = []
3   for i in xrange(1,len(poly)):
4     new.append(
5         float(i*poly[i]))
6   return new
7   if new==[]: return [0.0]
```

(b) An attempt to the derivatives problem with some statements in a wrong order.

Figure 1.2: Examples of student attempts requiring repair beyond expression modification.

Program repair A more guided feedback can be generated using *program repair*. Program repair, given a specification and a program that does not satisfy it, searches for a (minimal) set of modifications to the program, such that the modified program satisfies the specification. The modifications could, for example, be (we discuss them on an example below):

- to modify a single (sub-) expression to another expression,
- to add or remove a statement (or block of statements), and
- to change the order of statements.

Example 1.4. *In Example 1.3 we discussed the first type of a modification, where an expression is replaced by another expression.*

Figure 1.2 shows two (incorrect) attempts to the derivative problem [11]*. We discuss possible repairs (modifications).*

A possible modification in the (a) case is to add the missing `return new` *statement, after the statement at the lines 6-7. This is the second type of a modification where a new statement is added to the program.*

A possible modification in the (b) case is to swap the order of the two statements at the lines 6 and 7, respectively [12]*. This is the third type of a modification where the order of statements is swapped.*

The idea to use program repair for feedback generation was pioneered by AutoGrader [SGSL13]. AutoGrader takes as input a reference solution and a list of potential corrections (in the form of expression rewrite rules), both provided by a teacher, and searches for a set of

[11] These two attempts are not real student attempts, but manually created variations of the attempt in Figure 1.1 used for demonstration.

[12] We point out that in this particular case a simple dead-code analysis would infer that the statement at the line 7 is unreachable; however, such a modification can also be needed in cases when a simple analysis cannot infer this.

minimal corrections using a SAT [13]-based technique. However, AutoGrader is limited in a couple of ways:

- due to a huge search space it can generate only up to 4 modifications, beyond which its performance degrades significantly,
- it only modifies program expressions as opposed to performing other kinds of modifications such as inserting new statements, changing order of statements, or handling larger conceptual errors;
- it requires a domain-expert to provide a list of potential corrections, for each assignment separately.

The general purpose program repair techniques are also not fit for this task: the techniques discussed in Goues et al. [Gou+15] on the IntroClass benchmark, either repair a small number of defects (usually <50%) or take a long time (i.e., over one minute).

Performance feedback Most of research in this area has concentrated on functional correctness of student attempts, but research on non-functional properties (especially performance) has seen little attention. More precisely, prior to this work, there was little understanding of the types of performance mistakes that students make in introductory programming. We also point out the last two mentioned approaches (failing test-cases and repairs) would hardly be of any help for performance issues in student attempts.

A way to detect performance problems would be by using a (static) *bound-analysis* technique (e.g., Sinn, Zuleger, and Veith [SZV14]); these techniques compute the worst-case asymptotic complexity of a program. However, these techniques could only detect a problem (for example, if we new that the optimal solution is in $\mathcal{O}(n)$, and the student's attempt is in $\mathcal{O}(n^2)$), but not point the student towards the cause of the problem.

1.2 Aim of the thesis

The primary goal of these thesis is to develop new feedback generation techniques, both for performance and functional aspects of student programs. We next discuss the identified challenges towards this goal.

The first challenge that we face is:

Very little research has been done in the area of performance-related feedback and hence it is not clear what kind of performance-related mistakes students make in introductory programming, and consequently it is not clear how to provide feedback.

Based on the discussion in the previous section we identify essential desirable characteristics of a feedback generation technique:

[13]SAT or Boolean satisfiability problem.

1. *Automation* (or semi-automation): feedback should be generated using as little human effort as possible.
2. *Performance*: feedback should be generated in order of seconds to enable interactive learning.
3. *Correctness*: feedback should be generated by using sound techniques that guarantee correctness.
4. *Exhaustiveness*: feedback should be generated for most of the cases in introductory programming.
5. *Usefulness*: feedback should help students to understand the mistakes they make and how to make progress on one hand, and on the other hand to help teachers reduce the work they need to put on managing a large classroom.

We point out that every method discussed in the previous section fails on several of these characteristics:

- The peer-feedback approach, as discussed above, fails on several characteristics: automation, performance, correctness, and usefulness.
- The failing-tests approach is automated, fast, and exhaustive, however theoretically it can be unsound [14], and it does not provide enough guidance for students.
- The repair-based techniques are (mostly) automated, sound, and useful [15], however, they are, as discussed, often slow and can handle only a small portion of cases.

Hence, the second challenge we face is:

How to combine automation, speed, correctness, exhaustiveness, and usefulness in an approach to feedback generation?

In this thesis we take an approach of *dynamic program analysis*, similar to testing. We believe that dynamic analysis approach, although theoretically unsound, works well in practice for introductory programming. In other words, we aim to show that this kind of approach enables all of the above mentioned characteristics (with the obvious relaxation of the correctness criterion). However, we believe that the approach, although based on dynamic analysis, should have strong mathematical foundation.

Finally, we concretize and summarize the discussion from above:

- We aim to study and understand performance-related issues in introductory programming.

[14]It can show that an incorrect program is correct if some test-cases are missing.

[15]Although we point out that pedagogical issues around repair-based feedback are still an area of research.

- We aim to develop methods for performance and functional feedback generation in introductory programming that are automated, fast, correct, exhaustive, and useful. To succeed in this we want the develop methods that have:
  - a rigorous mathematical foundation; and
  - a practical implementation with an experimental evidence showing the desired characteristics.

1.3 Methodology

Next we discuss the research methodology used in this thesis.

The methodology, following the common practice in *formal methods* and *programming languages* research, and guided by the above stated aims, includes the following steps:

(1) Study of existing student programs to understand the problem.

(2) Formulation of an approach to solve the problem.

(3) Development of a rigorous mathematical framework underlying the approach.

(4) Implementation of the proposed approach in a practical tool.

(5) Experimental evaluation of the proposed approach.

We describe each of the steps in more detail.

Study of existing student programs The first step is to understand the problem on real-world examples of the problem. In our case that means studying a large number of student programs. This step was especially important for understanding how to provide performance-related feedback, because, as it was already mentioned, this area was not well understood or researched.

Formulation of an approach After understanding the problem on practical examples, next step is to come up with an idea on how to solve it. This step includes studying existing approaches for the problem at hand and similar problems, to understand what has been done and why existing approaches cannot solve the problem.

Development of a rigorous mathematical framework When we have an idea how to solve the problem, we mathematically model the approach. Having a mathematical framework is important for couple of reasons:

1. It gives certain guarantees about the soundness of the approach.

2. It often reveals that a problem being solved can be modeled as an already-known problem that might have well-known solution strategies; if not for the whole problem, then for some particular instances of interest.

3. A mathematical formulation is often simpler than the practical implementation, due to a higher level of abstraction, which often leads to insights how to simplify or improve both the ideas of the approach and the practical tools.

Implementation of the proposed approach This step is vital in practical research as we cannot do experimental evaluation without it. Further, it can serve to quickly check the viability of the approach. This step sometimes comes before mathematical formulation.

Experimental evaluation The final step is to evaluate how the proposed approach, that is, its implementation, performs on benchmarks of interest; e.g., by:

- Measuring how many tests in a benchmark the approach can handle.
- Measuring quantitative characteristics on each test (e.g., time required to handle a test).
- Performing manual qualitative assessments on a subset of tests.
- Performing a user study with human participants.

If a competitive approach for the problem already exists, the goal is to show that the proposed approach is better than the existing approach (using the above mentioned metrics).

We point out that these steps are often intertwined and that a process is refined until we are satisfied with the result. That is, often observations from the experimental evaluation, especially on the cases where the tool does not perform well, enables insights on improvements of the approach. However, one needs to be careful to avoid overfitting of the solution for one particular benchmark. This can be done by splitting the benchmark data into two sets (similar to practices in machine learning): one for improving the approach, and one for the evaluation.

1.4 Overview of the Thesis

We next give a high-level overview of the thesis, explained on some examples.

1.4.1 Study of Correct Student Solutions

We performed a manual study on a large number of correct student solutions, to understand the performance problems faced by students in introductory programming classes. We next discuss the detail of this study.

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length)
    return false;
  foreach (char c in t.ToCharArray()) {
    int index = s.IndexOf(c);
    if (index < 0) return false;
    s = s.Remove(index, 1);
  }
  return true;
}
```

(b) Removing / Library (**R1**)

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length)
    return false;
  char[] sa = s.ToCharArray();
  char[] ta = t.ToCharArray();
  for (int j=0; j < sa.Length; j++) {
    for (int i=0; i<sa.Length - 1;i++) {
      if (sa[i]<sa[i+1]){
        char temp=sa[i];
        sa[i]=sa[i+1];
        sa[i+1]=temp;
      }
      if (ta[i]<ta[i+1]){
        char temp=ta[i];
        ta[i] = ta[i+1];
        ta[i+1] = temp;
      }
    }
  }
  for (int k = 0; k < sa.Length; k++) {
    if (sa[k] != ta[k]) return false;
  }
  return true;
}
```

(a) Sorting / Bubble (**S1**)

```
bool Puzzle(string s, string t) {
  if (s.Length !=t.Length)
    return false;
  int [] cs=new int [256];
  int [] ct=new int [256];
  for(int i=0;i<s.Length;i++)
    cs[(int) s[i]]++;
  for(int i=0;i<t.Length;++i)
    ct[(int) t[i]]++;
  for (int i=0;i<256;i++)
    if(cs[i] != ct[i]) return false;
  return true;
}
```

(c) Efficient / Counting (**E1**)

Figure 1.3: Different algorithmic strategies for the anagram problem.

Student solutions We studied a large number of correct student solutions on the Pex4Fun platform, where students can solve different programming problems. The correct student solutions came from two sources: [16]

- 3 programming problems that already existed on Pex4Fun. These 3 problems were solved by various users of Pex4Fun.
- 21 programming problems that we created on Pex4Fun for our study. These 21 problems were assigned as a homework and solved by second-year undergraduate computer science students.

We point out that in the homework problems we encouraged students to pay attention to asymptotic worst-case complexity of their code, and gave additional points for asymptotically efficient solutions. In other words, we wanted to see the real performance problems that students in introductory programming course face, to understand how to provide feedback.

[16]The problems, data and experimental setting are explained in more detail in Section 3.7.

Study observations The main observations from this study are:

1. There are different *algorithmic strategies*, with varying level of efficiency, for solving a given problem. Algorithmic strategies capture the global high-level insight of a solution to a programming problem, while also defining key performance characteristics of the solution. Different strategies merit different feedback.

2. The same algorithmic strategy can be implemented in *many different ways*. These differences originate from local low-level implementation choices and are not relevant for reporting feedback on the student program.

We illustrate these points on some student solutions [17] to the *anagram problem* (already described in Example 1.1 as the requirement *F1*).

We first give examples of different algorithmic strategies. We point out that we manually determined these different strategies and solutions that belong to them.

Example 1.5 (Different algorithmic strategies)**.** *We have already given the student solution **C1** (in Figure 1.1); further examples of student solutions (**S1**, **R1** and **E1**) that employ different algorithmic strategies are given in Figure 1.3. The solutions **C1**, **S1**, and **R1** all employ inefficient strategies, since their complexities are in $\mathcal{O}(n^2)$. However, each of these solutions merits a different feedback. The solution **E1** employs an efficient strategy, since its complexity is in $\mathcal{O}(n)$. Below we explain these different strategies, and the corresponding feedback.*

*The solution **C1** implements the counting strategy. The counting strategy is to iterate over one of the input strings, and for each character in that string count and compare the number of occurrences of that character in both strings. An appropriate feedback in this case might be: "Calculate the number of characters in each string in a preprocessing phase, instead of in each iteration of the main loop".*

*The solution **S1** implements the sorting strategy. The sorting strategy is to sort both of the strings and check if they are equal after sorting. An appropriate feedback in this case might be: "Instead of sorting the input strings, compare the number of character occurrences in each string".*

*The solution **R1** implements the removing strategy. The removing strategy is to iterate over one of the input strings, and remove the corresponding character from the other string. An appropriate feedback in this case might be: "Use a more efficient data-structure to remove characters".*

*The solution **E1** implements the efficient counting strategy. The efficient counting strategy is to collect the number of occurrences of all characters from both strings, and then check that each character occurs the same number of times in both strings. An appropriate feedback in this case might be: "The code is asymptotically efficient".*

```
1 bool Puzzle(string s, string t) {
2   if (s.Length != t.Length)
3     return false;
4   else
5     return s.All(c =>
6       s.Where(c2 => c2 == c).Count() ==
7       t.Where(c2 => c2 == c).Count()
8       );
9 }
```

(a) Counting / Library (**C2**)

```
 1 bool Puzzle(string s, string t) {
 2   if(s.Length != t.Length)
 3     return false;
 4   foreach (var item in s) {
 5     if(s.Split(item).Length
 6         != t.Split(item).Length)
 7     return false;
 8   }
 9   return true;
10 }
```

(b) Counting / Split (**C3**)

Figure 1.4: Different implementations of the counting strategy.

Next, we give examples of different implementations of the above discussed algorithmic strategies.

Example 1.6 (Implementation details)**.** *Each of the strategies discussed above can be implemented in syntactically very different ways.*

For example, Figure 1.4 shows two additional implementations of the counting strategy, ***C2*** *and* ***C3****. The implementation* ***C1*** *manually implements counting function* `countChars` *at lines 12-20, while the implementations* ***C2*** *and* ***C3*** *count characters using built-in functions, at lines 6-7 and 5-6, respectively. However, the implementations* ***C2*** *and* ***C3*** *merit the same feedback as the implementation* ***C1*** *(discussed in the previous example).*

Similarly, Figure 1.5 shows two additional implementations for each of the sorting and removing strategies. Again, the implementations ***S2*** *and* ***S3*** *merit the same feedback as the implementation* ***S1****, and the implementations* ***R2*** *and* ***R3*** *merit the same feedback as the implementation* ***R1****.*

Thus, the take-away from the study is: *Given some strategy, all implementations of this strategy merit the same feedback. Therefore, to provide meaningful feedback to a student, it is important to identify what algorithmic strategy was used in the student's program, while ignoring the implementation details.*

Key Values

Hence, to provide an appropriate feedback, we need to distinguish different algorithmic strategies while ignoring implementation details. Our approach is based on the following key observation: different implementations of the same strategy generate the same *key values*, when executed on the same input.

We emphasize this on an example; a more detailed discussion follows in Section 2.1.

[17]We point out that these solutions are *real* student solutions from our study.

```
bool Puzzle(string s, string t) {
  var sa = s.ToCharArray();
  var ta = t.ToCharArray();
  Array.Sort(sa);
  Array.Sort(ta);
  return sa.SequenceEqual(ta);
}
```

(a) Sorting / Library (**S2**)

```
int BinarySearch(List<char> xs, char y) {
  int low = 0, high = xs.Count;
  while (low < high) {
    int mid = (high - low) / 2 + low;
    if (y < xs[mid]) high = mid;
    else if (y > xs[mid]) low = mid + 1;
    else return mid;
  }
  return low;
}
char[] Sort(string xs) {
  var res = new List<char>();
  foreach (var x in xs) {
    var pos = BinarySearch(res, x);
    res.Insert(pos, x);
  }
  return res.ToArray();
}
bool Puzzle(string s, string t) {
  return String.Join("", Sort(s))
    == String.Join("", Sort(t));
}
```

(b) Sorting / Binary Insertion (**S3**)

```
bool Puzzle(string s, string t) {
  return IsPermutation(s, t);
}
bool IsPermutation(String s, string t) {
  if (s == t) return true;
  if (s.Length != t.Length)
    return false;
  int index = t.IndexOf(s[0]);
  if (index == -1) return false;

  s = s.Substring(1);
  t = t.Remove(index, 1);

  return IsPermutation(s, t);
}
```

(c) Removing / Recursive (**R2**)

```
bool Puzzle(string s, string t) {
  char[] sc = s.ToCharArray();
  char[] tc = t.ToCharArray();
  Char c = '#';
  if(sc.Length != tc.Length)
    return false;
  for(int i=0;i<sc.Length;i++) {
    c = sc[i];
    for(int j=0;j<tc.Length;j++) {
      if(tc[j]==c){
        tc[j]='#';
        break;
      }
      if(j==tc.Length-1) {
        return false;
      }
    }
  }
  return true;
}
```

(d) Removing / Manual (**R3**)

Figure 1.5: Different implementations of the sorting and removing strategies.

Example 1.7. *The implementations **C1** and **C2** (discussed in the previous section), when executed on the input* `s="aba"` *and* `t="baa"`*, produce the following sequence of values on the underlined expressions:*

$$(a, b, a, 2, b, a, a, 2, a, b, a, 1, b, a, a, 1, a, b, a, 2, b, a, a, 2)$$

```
void Puzzle(string s, string t) {
  for (int i = 0; i < s.Length; ++i) {
    int cnt1 = 0, cnt2 = 0;
    for (int j = 0; j < s.Length; ++j) {
      observe(s[j]);
      if (s[j] == s[i]) {
        cnt1++;
      }
    }
    observe(cnt1);
    for (int j = 0; j < t.Length; ++j) {
      observe(t[j]);
      if (t[j] == s[i]) {
        cnt2++;
      }
    }
    observe(cnt2);
  }
}
```

(a) Counting strategy **CS**

```
void Puzzle(string s, string t) {
  char[] ca = s.ToCharArray();
  Array.Sort(ca);
  observe(ca);
}
```

(b) Sorting strategy **SS**

Figure 1.6: Examples of specifications.

*Note: Underlined expressions are at lines 5, 6, and 15 in **C1**, and at lines 6 and 7 in **C2**.*

1.4.2 Performance Feedback

We next discuss our approach for providing performance feedback, based on the key-value idea discussed above:

- The teacher specifies an algorithmic strategy by simply annotating (at the source code level) certain key values computed by a sample program (that implements the corresponding strategy) using a new language construct, called **`observe`**. We call this sample program a *specification*.
- We determine that a student's implementation implements some strategy if it *matches* the teacher's specification of this strategy: the implementation matches then specification if they compute the same (key-) values, at corresponding program locations, in the same order.

We discuss this on some examples; a detailed discussion, together with a methodology for providing feedback in a MOOC-size class, follows in Chapter 3.

Example 1.8. *Figure 1.6 gives two examples of specifications, **CS** and **SS**, for the counting and sorting strategies, respectively.*

*In the specification **CS**, for the counting strategy, the teacher observes:*

```
def computeDeriv(poly):
  result = []
  for e in range(1, len(poly)):
    result.append(float(poly[e]*e))
  if result == []:
    return [0.0]
  else:
    return result
```

(a) Correct solution **D1**.

```
def computeDeriv(poly):
  deriv = []
  for i in xrange(1,len(poly)):
    deriv+=[float(i)*poly[i]]

  if len(deriv)==0:
    return [0.0]
  return deriv
```

(b) Correct solution **D2**.

Figure 1.7: Examples of correct student solutions to the derivative problem.

```
def computeDeriv(poly):
  result = []
  for i in range(len(poly)):
    result[i]=float((i)*poly[i])
  return result
```

Figure 1.8: An additional incorrect student attempt (**I2**) to the derivative problem.

- *The characters being iterated over by the two **`observe`** statements at the lines 5 and 12.*
- *The results of counting the characters by the two **`observe`** statements at the lines 10 and 17.*

*In the specification **SS**, for the sorting strategy, the teacher observes the sorted input string by the **`observe`** statement at the line 4.*

*The implementations **C1** and **C2** match the specification **CS**, while the implementations **S1** and **S2** match the specification **SS**.*

Hence, these implementations are given feedback based on which specification they match, as discussed in Example 1.5.

1.4.3 Functional Feedback

Our approach to functional feedback is based on program repair (as discussed in Section 1.1); that is, we repair an incorrect student program to provide feedback to the student.

The repair algorithm uses existing correct student solutions (wisdom of the crowd) to repair incorrect student attempts.

The algorithm is based on a notion of program matching, similar to the one discussed for the performance matching (see below): given a correct solution and an incorrect attempt, the algorithm finds minimal set of modifications such that the repaired incorrect attempt matches the correct solution; further, the algorithm uses expressions from the correct solution to repair the incorrect attempt.

1. In return statement at line 7, change `0.0` to `[0.0]`.

(a) A repair for **I1**.

1. In iterator expression at line 3, change `range(len(poly))` to `range(1, len(poly))`.
2. In assignment at line 4, change `result[i]=float(i*poly[i])` to `result.append(float(i*poly[i]))`.
3. In return statement at line 5, change `result` to `result or [0.0]`.

(b) A repair for **I2**.

Figure 1.9: Examples of repairs for the incorrect student attempts.

The matching notion used in the algorithm is similar to the matching notion discussed for performance feedback: we say that two programs match if there is a bijective relation between the variables of two programs, such that related variables take the same values, during the execution of the programs on the same inputs. In Chapter 2 we develop a generalized version of these matching notions.

The overall approach consists of two main steps:

(I) We first cluster the correct solutions in equivalence classes, based on the notion of matching. This has a twofold purpose:

- It reduces the number of correct solutions that the repair algorithm needs to consider, by grouping together equivalent solutions.
- It collects different correct expressions from solutions in the same cluster; this is useful for diversity of repairs.

(II) Given an incorrect student attempt, the repair algorithm generates a repair for each cluster separately, and then chooses the minimal repair among them.

We now discuss this on some examples; the detailed discussion follows in Chapter 4.

Example 1.9. *Figure 1.7 shows two examples of correct student attempts to the derivative problem,* ***D1*** *and* ***D2****. These two correct solutions match because there is a bijective variable relation, such that related variables take the same values during the execution on the same inputs:* [18]

$$poly \mapsto poly, deriv \mapsto result, i \mapsto e$$

[18]This variable relation includes more variables (as discussed later in Chapter 4), however, these are sufficient to understand the example.

*Hence, the clustering step determines that **D1** and **D2** belong to the same cluster (let us call it $\mathcal{C}$).*

*In Figure 1.1 (b) we have shown the incorrect attempt **I1**, and Figure 1.8 shows the incorrect attempt **I2** (both to the derivatives problem).*

*Figure 1.9 shows repairs for the attempts **I1** and **I2**. Both of these repairs were generated using the cluster of correct solutions $\mathcal{C}$ (that contains **D1** and **D2**):*

- *The repair for **I1** was generated using the expression `return [0.0]` from line 7 of the correct solution **D2**.*
- *The repair for **I2** was generated using the expression `range(1, len(poly))` from line 3 of the correct solution **D1** and from correct expressions from other correct solutions in the cluster $\mathcal{C}$ (not shown here).*

1.5 Contributions

The main technical contributions of this thesis are published in two peer-reviewed conference papers:

(1) "*Feedback Generation for Performance Problems in Introductory Programming Assignments*" [GRZ14] - published at *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), Hong Kong, China.*

- This paper describes a novel approach for providing performance feedback on introductory programming assignments.

(2) "*Automated Clustering and Program Repair for Introductory Programming Assignments*" [GRZ18] - published at *39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018) Philadelphia, PA, USA.*

- This paper describes a novel algorithm for providing repair-based feedback on functional correctness of introductory programming assignments.

The approaches described in both papers were implemented in the following publicly available tools:

(1) Observer [19] - a tool for providing performance feedback on introductory programming assignments.

(2) Clara [20] - a tool for providing repair-based feedback on functional correctness of introductory programming assignments.

[19]`http://forsyte.at/static/people/radicek/fse14/`
[20]`https://github.com/iradicek/clara`

In this thesis we make the following contributions:

1. We study a large number of student attempts to understand the performance mistakes made in introductory programming. The result of this study is a novel idea of *algorithmic strategies*.

2. Based on the observations from the study we develop a novel dynamic relational program analysis, which is the core of the two proposed feedback generation approaches.

3. We propose a novel semi-automated approach for generating performance feedback in introductory programming. The approach consists of:

   (i) A new language construct, called **`observe`**, that allows the teacher to describe algorithmic strategies.

   (ii) A novel algorithm for deciding whether a student's implementation matches a teacher's specification.

4. We propose a novel fully-automated approach for generating functional feedback in introductory programming. The approach is based on a novel program repair algorithm that utilizes already existing student solutions.

5. We perform experimental evaluations of both of the approaches on a large number of real-world student programs and on a user study. We show that our approaches:

   (i) Generate feedback in almost all cases.

   (ii) Generate useful feedback and significantly reduce the required teacher effort.

   (iii) Generate feedback in order of seconds, hence making them suitable for an interactive teaching setting like MOOC.

1.6 Structure of the Thesis

The rest of this thesis is structured as follows.

In Chapter 2 we first discuss the notion of key values (Section 2.1) and define the core version (Section 2.2) of our dynamic relational program analysis (which is inspired by the notion of key values). Next, we describe various generalizations of the proposed dynamic program analysis (Section 2.3). The proposed analysis is the core ingredient used for providing both performance and functional feedback; that is, the common ideas of the two conference papers (mentioned in the previous section) are factorized in this chapter.

In Chapter 3 we describe our approach for providing performance feedback, and in Chapter 4 we describe our approach for providing functional (repair-based) feedback.

Both of these two chapters, describing the two approaches, follows the same structure:

- We first give a high-level overview of the approach, explained on examples (Section 3.1 and Section 4.1).
- We continue by giving preliminaries and the language model, required to formally describe our approach and the algorithms (Section 3.2 and Section 4.2). This is followed by the formal discussion of the core algorithms (Section 3.3 and Section 4.3), and by describing some useful extensions to the core algorithms (Section 3.4 and Section 4.4).
- Then we describe the methodology of the approach, that is, how is the approach used in a real classroom or MOOC setting (Section 3.5 and Section 4.5). Finally, we discuss the implementation (Section 3.6 and Section 4.6) and the experimental evaluation of the approach (Section 3.7 and Section 4.7).

In the rest of the thesis, we give an overview of the related work in Chapter 5, and in Chapter 6 we discuss in more detail the contributions of the thesis and some possible directions for future work.

CHAPTER 2

Dynamic Relational Analysis

In this section we discuss a program analysis that is the core ingredient for both performance and functional feedback generation approaches described later.

We point out that the core of this analysis is inspired by the manual code study discussed in Section 1.4.1. Hence, we restate the main observations of this study here:

1. There are different *algorithmic strategies*, with varying level of efficiency, for solving a given problem. Different strategies merit different feedback.
2. The same algorithmic strategy can be implemented in *many different ways*. These differences are not relevant for reporting feedback on the student program.

This chapter starts with the discussion of the basic version of our analysis, in Section 2.1 and Section 2.2, followed by a discussion of various generalizations of it, in Section 2.3.

2.1 Key Values

Following the observations from the study, the main challenge is:
How can we identify the algorithmic strategy used in a student program, while ignoring all the low-level implementation details that are not relevant for feedback?

Key values Our key insight is that different implementations of the same algorithmic strategy generate the same *key values* during their execution on the same input.

We illustrate this on an example.

Example 2.1. *The implementations **C1** and **C2** (Figure 1.1 and Figure 1.4), when executed on the input* `s="aba"` *and* `t="baa"`*, produce the following sequence of values* *on the underlined expressions:*

$$(a, b, a, 2, b, a, a, 2, a, b, a, 1, b, a, a, 1, a, b, a, 2, b, a, a, 2)$$

*Note: Underlined expressions are at lines 5, 6, and 15 in **C1**, and at lines 6 and 7 in **C2**.*

So how can we use this insight to decide whether a program is an implementation of some strategy? In other words:

1. How do we determine what are the key values that define a strategy (on some input)?
2. How do we determine if a student program computes those values as well?

We defer the first question for later and for now assume that there is an oracle program that provides key values for a strategy. We discuss this further in Chapter 3.

Hence, for the second question we assume that we are given the key values of some strategy (for some input). However, merely observing that a program computes the same set of values somewhere in the program will not give us very strong guarantees; we could imagine a myriad of different programs that compute the same set of values.

Nonetheless, if we study the previous example closer we observe the following:

- The values are computed in exactly the same order.
- The matching values are computed at corresponding locations in two programs.

Hence, we want the following:

- For each key value we also associate a program location where the value is computed.
- For each program location in the key value sequence there is a corresponding location in the implementation that produces the same values.
- The values are computed in the same order.

We discuss this on an example.

Example 2.2. *Let us assume that for the counting strategy, on the input* `s="aba"` *and* `t="baa"`*, we are given the following sequence of key values with associated locations:*

$$((\ell_1, a), (\ell_1, b), (\ell_1, a), (\ell_2, 2), (\ell_3, b), (\ell_3, a), (\ell_3, a), (\ell_4, 2), (\ell_1, a), (\ell_1, b), (\ell_1, a), (\ell_2, 1), (\ell_3, b), (\ell_3, a), (\ell_3, a), (\ell_4, 1), (\ell_1, a), (\ell_1, b), (\ell_1, a), (\ell_2, 2), (\ell_3, b), (\ell_3, a), (\ell_3, a), (\ell_4, 2))$$

Where ℓ_1 - ℓ_4 are (abstract) program locations representing the following in the counting strategy.

- *ℓ_1 is the location where the characters of the first string (s) are iterated;*
- *ℓ_2 is the location where the number (count) of characters in s is computed;*
- *ℓ_3 is the location where the characters of the first string (t) are iterated; and*
- *ℓ_4 is the location where the number (count) of characters in t is computed;*

*The corresponding program locations in **C1** and **C2** are the underlined expressions (as discussed in the previous example).* [1]

This analysis is inspired by the notion of a *simulation relation* [Mil71], adapted for a dynamic program analysis: whereas a simulation relation establishes that a program P produces exactly the same values as program Q at corresponding program locations for all inputs, we are interested only in a fixed finite set of inputs. Therefore, this notion could be seen as a *dynamic simulation relation*, to stress that we use dynamic program analysis.

Later in this chapter we generalize the key-value approach discussed here in several ways. However, the core ideas in these generalizations remain: a *dynamic program analysis* and a simultaneous analysis of executions of *two programs*. The latter is usually called *relational program analysis*, as it relates executions of two program, opposed to the usual program analysis that reasons about executions of a single program [2]. Hence, our approach can be seen as a dynamic relational analysis.

2.2 Trace Embedding

In this section we formally define the key-value analysis described in the previous section; we call this analysis *Trace Embedding*.

We start by stating some preliminary definitions.

Definition 2.3 (Program location, computation value, computation trace)**.** *Let Loc be a set of program locations, and let Val be a set of computation values.*

A computation trace γ over some set of locations Loc is a finite sequence of location-value pairs $(Loc \times Val)^*$. Let Γ_{Loc} be the set of all traces over Loc.

Given some trace $\gamma \in \Gamma_{Loc}$, and some set of locations $Loc_2 \subseteq Loc$, let $\gamma|_{Loc_2}$ denote a sequence obtained from γ by deleting all pairs (ℓ, val), where $\ell \notin Loc_2$.

[1]Note that in **C1** there are only three underlined expressions, while in the key-value sequence there are four locations. However, we consider the underlined expression at line 15 as two distinct program locations, depending on whether the function `countChars` is called from line 5 or 6. These details are further discussed in Chapter 3.

[2]Actually, relational program analysis reasons either about executions of two programs, or two executions of the same program. In functional programming, where there is usually no distinction between programs and data, these can be seen as equivalent.

```
def subsequence($\gamma_1$, $\gamma_2$):
    n = length($\gamma_1$)
    m = length($\gamma_2$)
    if m < n:
        return False
    $k_1$ = 1
    for all $1 \leq k_2 \leq m$:
        $(\ell_1, val_1)$ = $\gamma_1[k_1]$ # $k_1^{\text{th}}$ element of $\gamma_1$
        $(\ell_2, val_2)$ = $\gamma_2[k_2]$ # $k_2^{\text{th}}$ element of $\gamma_2$
        if $\ell_1 = \ell_2$ and $val_1 = val_2$:
            if $k_1$ = $n$:
                return True
            $k_1$ = $k_1$ + 1
    return False
```

Figure 2.1: Algorithm for the subsequence problem.

We do not define what exactly program locations or computation values are, as this is not necessary to define our analysis. In practice program locations could be assigned, for example, to each statement or each basic block; similarly computation values could consists of all constant values in some language. Further, we do not discuss here how a computation trace is obtained from a program, or what a program model is. We will define all of these more precisely in Chapter 3 and Chapter 4; as we will see these notions should be chosen according to specific analysis needs.

For the remaining of this chapter we assume some fixed set of computation values *Val*.

Definition 2.4 (Subsequence). *Let $\gamma_1 = (\ell_{1,1}, val_{1,1}) \cdots (\ell_{1,n}, val_{1,n})$, and $\gamma_2 = (\ell_{2,1}, val_{2,1}) \cdots (\ell_{2,m}, val_{2,m})$ be two computation traces over some set of locations Loc.*

Then, γ_1 is a subsequence of γ_2, written $\gamma_1 \sqsubseteq \gamma_2$, when:

- *There are indices $1 \leq k_1 < k_2 < \cdots k_n \leq m$, such that for all $1 \leq i \leq n$ we have $\ell_{1,i} = \ell_{2,k_i}$ and $val_{1,i} = val_{2,k_i}$.*

Figure 2.1 gives an algorithm for deciding the subsequence relation between traces γ_1 and γ_2. Note that the operation performs *at most m* (= length of γ_2) equality checks.

Definition 2.5 (Mapping function). *Given two sets of locations Loc_1 and Loc_2, an injective function $\pi : Loc_1 \to Loc_2$ is called a mapping function.*

Mapping function lifted to $\pi : \Gamma_{Loc_1} \to \Gamma_{Loc_2}$ is defined by applying it to every location:

$$\pi((\ell_1, val_1) \cdots (\ell_n, val_n)) = (\pi(\ell_1), val_1) \cdots (\pi(\ell_n), val_n)$$

Definition 2.6 (Trace Embedding). *A computation trace $\gamma_1 \in \Gamma_{Loc_1}$ can be embedded in a trace $\gamma_2 \in \Gamma_{Loc_2}$, if there exists $\pi : Loc_1 \to Loc_2$ such that $\pi(\gamma_1) \sqsubseteq \gamma_2$, written $\gamma_1 \sqsubseteq^{\pi} \gamma_2$. The mapping π is then called an embedding witness.*

Executing a program on a set of inputs I results in a set of traces, one for each input $i \in I$. A set of traces $(\gamma_{1,i})_{i\in I}$ can be embedded in a set of traces $(\gamma_{2,i})_{i\in I}$ by π if and only if $\gamma_{1,i} \sqsubseteq^\pi \gamma_{2,i}$ for all $i \in I$.

The Trace Embedding *problem consists of finding an embedding witness, between two sets of traces.*

Next, we illustrate the trace embedding problem on an example.

Example 2.7. *Consider the following traces*

$$\begin{aligned}\gamma_1 &= \qquad\quad (\ell_1, 1) \cdot (\ell_2, 1) \qquad\qquad \cdot (\ell_2, 2) \cdot (\ell_1, 2) \qquad\qquad \cdot (\ell_2, 3)\\ \gamma_2 &= (\ell_3, 0) \cdot (\ell_4, 1) \cdot (\ell_5, 1) \cdot (\ell_3, 3) \cdot (\ell_5, 2) \cdot (\ell_4, 2) \cdot (\ell_4, 3) \cdot (\ell_5, 3)\end{aligned}$$

and let $\pi = \{\ell_1 \mapsto \ell_4, \ell_2 \mapsto \ell_5\}$ be a mapping function.

Then we have $\gamma_1 \sqsubseteq^\pi \gamma_2$. We point out that the notion of trace embedding allows the location ℓ_3 to be not mapped to any location of the trace γ_1, and that there is the element $(\ell_4, 3)$ in γ_2 that has no corresponding element in the trace γ_1.

Complexity of Trace Embedding Assuming that equality checks can be done in polynomial time, Trace Embedding is in NP: Given some embedding witness π it is easy to check if $\gamma_{1,i} \sqsubseteq^\pi \gamma_{2,i}$, for all $i \in I$. As discussed above, this can be checked in linear number of (polynomial) equality checks. However, it turns out that Trace Embedding is NP-complete. This holds *even* for a *singleton* input I and a *singleton* computation domain *Val*.

Theorem 2.8 (Trace Embedding NP-completeness)**.** *The Trace Embedding problem is NP-complete, assuming equality checks can be done in polynomial time.*

Proof. In order to show NP-hardness we reduce Permutation Pattern [BBL98] to Trace Embedding. We point out that Permutation Pattern is NP-complete.

First, we formally define Permutation Pattern. Let n, k be positive integers with $k \leq n$. Let σ be a permutation of $\{1, \cdots, n\}$ and let τ be a permutation of $\{1, \cdots, k\}$. We say τ *occurs* in σ, if there is an injective function $\pi : \{1, \cdots, k\} \to \{1, \cdots, n\}$ such that π is monotone, i.e., for all $1 \leq r < s \leq k$ we have $\pi(r) < \pi(s)$ and $\pi(\tau(1)) \cdots \pi(\tau(k))$ is a subsequence of $\sigma(1) \cdot \sigma(2) \cdots \sigma(n)$. *Permutation Pattern* is the problem of deciding whether τ occurs in σ.

We now give the reduction of Permutation Pattern to Trace Embedding. We will construct two traces γ_1 and γ_2 over a singleton computation domain *Val*, and over the sets of locations $Loc_1 = \{1, \ldots, k\}$ and $Loc_2 = \{1, \ldots, n\}$. Because *Val* is singleton, we can ignore values in the rest of the proof. We set $\gamma_1 = 1 \cdot 2 \cdots k \cdot \tau(1) \cdot \tau(2) \cdots \tau(k)$ and $\gamma_2 = 1 \cdot 2 \cdots n \cdot \sigma(1) \cdot \sigma(2) \cdots \sigma(n)$. We now show that τ occurs in σ iff there is an

```
 1 def EMBED(($(\gamma_{1,i})_{i\in I}$, $(\gamma_{2,i})_{i\in I}$, $Loc_1$, $Loc_2$):
 2     G = $Loc_1 \times Loc_2$
 3     for all $\ell_1 \in Loc_1$, $\ell_2 \in Loc_2$:
 4         for all $i \in I$:
 5             if $\gamma_{1,i}|_{\{\ell_1\}} \not\sqsubseteq^{\{\ell_1 \mapsto \ell_2\}} \gamma_{2,i}|_{\{\ell_2\}}$:
 6                 G = G $\setminus\{(\ell_1, \ell_2)\}$
 7                 break
 8     for all $\pi \in$ MaximumBipartiteMatching(G):
 9         found = True
10         for all $i \in I$:
11             if $\gamma_{1,i} \not\sqsubseteq^{\pi} \gamma_{2,i}$:
12                 found = False
13                 break
14         if found:
15             return True
16     return False
```

Figure 2.2: Algorithm for the Trace Embedding problem.

injective function $\pi : Loc_1 \to Loc_2$ with $\gamma_1 \sqsubseteq^{\pi} \gamma_2$. We establish this equivalence by two observations: First, because every $i \in \{1, \cdots, k\}$ occurs exactly twice in γ_1 and γ_2 we have $1 \cdot 2 \cdots k \sqsubseteq^{\pi} 1 \cdot 2 \cdots n$ and $\tau(1) \cdot \tau(2) \cdots \tau(k) \sqsubseteq^{\pi} \sigma(1) \cdot \sigma(2) \cdots \sigma(n)$ iff $\gamma_1 \sqsubseteq^{\pi} \gamma_2$. Second, $1 \cdot 2 \cdots k \sqsubseteq^{\pi} 1 \cdot 2 \cdots n$ iff $\pi : Loc_1 \to Loc_2$ is monotone. $\square$

Algorithm Figure 2.2 shows our algorithm, Embed, for the Trace Embedding problem. A straightforward algorithmic solution for the Trace Embedding problem is to simply test all possible mapping functions. However, there is an *exponential number* of such mapping functions w.r.t. the cardinality of Loc_1 and Loc_2. This exponential blowup seems unavoidable as the combinatorial search space is responsible for the NP hardness. The *core element* of the algorithm is a pre-analysis that narrows down the space of possible mapping functions effectively.

We observe that if $\ell_2 = \pi(\ell_1)$ and $\gamma_1 \sqsubseteq^{\pi} \gamma_2$, then there exists a trace embedding restricted to locations ℓ_1 and ℓ_2; formally: $\gamma_1|_{\{\ell_1\}} \sqsubseteq^{\{\ell_1 \mapsto \ell_2\}} \gamma_2|\{\ell_2\}$. The algorithm uses this insight to create a (bipartite) graph $G \subseteq Loc_1 \times Loc_2$ of potential mapping pairs in lines 2-7. A pair of locations $(\ell_1, \ell_2) \in G$ is a *potential mapping pair* iff there exists a trace embedding restricted to locations ℓ_1 and ℓ_2 as described above.

The key idea in finding an embedding witness π is to construct a *maximum bipartite matching* [Uno97] in G. A maximum bipartite matching has an edge connecting every program location from Loc_1 to a distinct location in Loc_2 and thus gives rise to an injective function π. However, such an injective function π does not need to be an embedding witness, because, by observing only a single location pair at a time, it ignores the order of locations. Thus, for each maximum bipartite matching π the algorithm checks (in lines 8-15) if it is indeed an embedding witness.

The key strength of the algorithm is that it reduces the search space for possible embedding witnesses π. The experimental evidence shows that this approach significantly reduces the number of possible matchings and enables a very efficient algorithm in practice, as discussed in Section 3.7.

Finally, we discuss the algorithm on an example.

Example 2.9. *For this example we set the computation values to natural numbers, that is, $Val = \mathbb{N}$.*

Let

$$\begin{array}{rcl} Loc_1 & = & \{\ell_{1,1}, \ell_{1,2}, \ell_{1,3}, \ell_{1,4}\} \\ Loc_2 & = & \{\ell_{2,1}, \ell_{2,2}, \ell_{2,3}, \ell_{2,4}, \ell_{2,5}\} \\ Loc_3 & = & \{\ell_{3,1}, \ell_{3,2}, \ell_{3,3}, \ell_{3,4}\} \\ Loc_4 & = & \{\ell_{4,1}, \ell_{4,2}, \ell_{4,3}, \ell_{4,4}, \ell_{4,5}\} \end{array}$$

Consider the following traces, over the locations Loc_1, Loc_2, Loc_3, and Loc_4, respectively:

$$\begin{array}{rcl} \gamma_1 & = & (\ell_{1,1}, 5) \cdot (\ell_{1,2}, 0) \cdot (\ell_{1,3}, 0) \cdot (\ell_{1,3}, 1) \cdot (\ell_{1,2}, 1) \cdot (\ell_{1,3}, 0) \cdot (\ell_{1,2}, 2) \cdot (\ell_{1,3}, 0) \\ & & \cdot(\ell_{1,3}, 1) \cdot (\ell_{1,3}, 2) \cdot (\ell_{1,4}, 5) \\ \gamma_2 & = & (\ell_{2,1}, 4) \cdot (\ell_{2,2}, 0) \cdot (\ell_{2,3}, 0) \cdot (\ell_{2,3}, 1) \cdot (\ell_{2,2}, 1) \cdot (\ell_{2,3}, 0) \cdot (\ell_{2,3}, 1) \cdot (\ell_{2,2}, 2) \\ & & \cdot(\ell_{2,3}, 0) \cdot (\ell_{2,3}, 1) \cdot (\ell_{2,3}, 2) \cdot (\ell_{2,4}, 5) \cdot (\ell_{2,5}, 5) \\ \gamma_3 & = & (\ell_{3,1}, 4) \cdot (\ell_{3,2}, 0) \cdot (\ell_{3,3}, 0) \cdot (\ell_{3,3}, 1) \cdot (\ell_{3,2}, 1) \cdot (\ell_{3,3}, 0) \cdot (\ell_{3,3}, 1) \cdot (\ell_{3,2}, 2) \\ & & \cdot(\ell_{3,3}, 0) \cdot (\ell_{3,3}, 1) \cdot (\ell_{3,3}, 2) \cdot (\ell_{3,4}, 5) \\ \gamma_4 & = & (\ell_{4,1}, 5) \cdot (\ell_{4,2}, 0) \cdot (\ell_{4,3}, 0) \cdot (\ell_{4,3}, 1) \cdot (\ell_{4,2}, 1) \cdot (\ell_{4,3}, 0) \cdot (\ell_{4,3}, 1) \cdot (\ell_{4,2}, 2) \\ & & \cdot(\ell_{4,3}, 0) \cdot (\ell_{4,3}, 1) \cdot (\ell_{4,3}, 2) \cdot (\ell_{4,4}, 4) \cdot (\ell_{4,5}, 5) \end{array}$$

We first check whether the trace γ_1 can be embedded in the trace γ_2. [3] *In the first step of the algorithm we have the following:*

$$\begin{array}{ccc} \gamma_1|_{\{\ell_{1,1}\}} & \sqsubseteq^{\{\ell_{1,1} \mapsto \ell_{2,4}\}} & \gamma_2|_{\{\ell_{2,4}\}} \\ \gamma_1|_{\{\ell_{1,1}\}} & \sqsubseteq^{\{\ell_{1,1} \mapsto \ell_{2,5}\}} & \gamma_2|_{\{\ell_{2,5}\}} \\ \gamma_1|_{\{\ell_{1,2}\}} & \sqsubseteq^{\{\ell_{1,2} \mapsto \ell_{2,2}\}} & \gamma_2|_{\{\ell_{2,2}\}} \\ \gamma_1|_{\{\ell_{1,2}\}} & \sqsubseteq^{\{\ell_{1,2} \mapsto \ell_{2,3}\}} & \gamma_2|_{\{\ell_{2,3}\}} \\ \gamma_1|_{\{\ell_{1,3}\}} & \sqsubseteq^{\{\ell_{1,3} \mapsto \ell_{2,3}\}} & \gamma_2|_{\{\ell_{2,3}\}} \\ \gamma_1|_{\{\ell_{1,4}\}} & \sqsubseteq^{\{\ell_{1,4} \mapsto \ell_{2,4}\}} & \gamma_2|_{\{\ell_{2,4}\}} \\ \gamma_1|_{\{\ell_{1,4}\}} & \sqsubseteq^{\{\ell_{1,4} \mapsto \ell_{2,5}\}} & \gamma_2|_{\{\ell_{2,5}\}} \end{array}$$

Hence, the algorithms constructs the following graph of potential mapping pairs:

$$G_{1,2} = \{(\ell_{1,1}, \ell_{2,4}), (\ell_{1,1}, \ell_{2,5}), (\ell_{1,2}, \ell_{2,2}), (\ell_{1,2}, \ell_{2,3}), (\ell_{1,3}, \ell_{2,3}), (\ell_{1,4}, \ell_{2,4}), (\ell_{1,4}, \ell_{2,5})$$

From $G_{1,2}$ the algorithm obtains the following two maximum bipartite matchings:

$$\begin{array}{rcl} \pi_1 & = & \{\ell_{1,1} \mapsto \ell_{2,4}, \ell_{1,2} \mapsto \ell_{2,2}, \ell_{1,3} \mapsto \ell_{2,3}, \ell_{1,4} \mapsto \ell_{2,5}\} \\ \pi_2 & = & \{\ell_{1,1} \mapsto \ell_{2,5}, \ell_{1,2} \mapsto \ell_{2,2}, \ell_{1,3} \mapsto \ell_{2,3}, \ell_{1,4} \mapsto \ell_{2,4}\} \end{array}$$

[3] In this example we assume a single input, that is, singleton traces. This is done to make the presentation easier to follow. Analysis of two sets of traces (each for one input) is analogous.

However, we have $\gamma_1 \not\sqsubseteq^{\pi_1} \gamma_2$ and $\gamma_1 \not\sqsubseteq^{\pi_2} \gamma_2$, hence the algorithm concludes that γ_1 cannot be embedded in γ_2.

Next, we check whether the trace γ_1 can be embedded in the trace γ_3. In the first step the algorithm constructs the following graph of potential mapping pairs:

$$G_{1,3} = \{(\ell_{1,1}, \ell_{3,4}), (\ell_{1,2}, \ell_{3,2}), (\ell_{1,2}, \ell_{3,3}), (\ell_{1,3}, \ell_{3,3}), (\ell_{1,4}, \ell_{3,4})\}$$

There are not maximum bipartite matchings in $G_{1,3}$ because the locations $\ell_{1,1}$ and $\ell_{1,4}$ would both need to be mapped to the location $\ell_{3,4}$, and that mapping would not be injective. Hence, the algorithm concludes that γ_1 cannot be embedded in the trace γ_3.

Finally, we check whether the trace γ_1 can be embedded in the trace γ_4. In the first step the algorithm construct the following graph of potential mapping pairs:

$$G_{1,4} = \{(\ell_{1,1}, \ell_{4,1}), (\ell_{1,1}, \ell_{4,5}), (\ell_{1,2}, \ell_{4,2}), (\ell_{1,2}, \ell_{4,3}), (\ell_{1,3}, \ell_{4,3}), (\ell_{1,4}, \ell_{4,1}), (\ell_{1,4}, \ell_{4,5})\}$$

From $G_{1,4}$ the algorithm obtains the following two maximum bipartite matchings:

$$\begin{array}{rcl} \pi_3 & = & \{\ell_{1,1} \mapsto \ell_{4,5}, \ell_{1,2} \mapsto \ell_{4,2}, \ell_{1,3} \mapsto \ell_{4,3}, \ell_{1,4} \mapsto \ell_{4,1}\} \\ \pi_4 & = & \{\ell_{1,1} \mapsto \ell_{4,1}, \ell_{1,2} \mapsto \ell_{4,2}, \ell_{1,3} \mapsto \ell_{4,3}, \ell_{1,4} \mapsto \ell_{4,5}\} \end{array}$$

We have $\gamma_1 \not\sqsubseteq^{\pi_3} \gamma_4$, but $\gamma_1 \sqsubseteq^{\pi_4} \gamma_4$. Hence, the algorithm concludes that the trace γ_1 can be embedded in the trace γ_4.

2.3 Generalizations of Trace Embedding

Next we discuss generalizations of the Trace Embedding problem discussed in the previous section. These generalizations will be useful later in Chapter 3 and Chapter 4.

2.3.1 Trace Relations

In the previous section we defined the Trace Embedding problem using the subsequence relation (Definition 2.4) on two traces. Now we define two additional trace relations.

Full Subsequence Relation

Consider the following example.

Example 2.10. *Let*

$$\begin{array}{l} \gamma_1 = \qquad\qquad (\ell_1, val_2)\cdot \qquad\qquad\qquad\qquad\qquad (\ell_2, val_5) \cdot (\ell_1, val_6) \\ \gamma_2 = (\ell_2, val_1)\cdot(\ell_1, val_2) \cdot (\ell_2, val_3) \cdot (\ell_3, val_4)\cdot(\ell_2, val_5) \cdot (\ell_1, val_6) \end{array}$$

Then we have $\gamma_1 \sqsubseteq \gamma_2$.

We see that the subsequence relation, as defined in the previous section, allows that γ_2 has more elements over the locations present in both the traces.

More precisely, the trace γ_2 contains two additional elements ((ℓ_2, val_1) and (ℓ_2, val_3)) over the location ℓ_2 than the trace γ_2.[4]

However, we could be interested in a relation that restricts traces to have the same number of elements over all locations present in the trace γ_1, and allow extra elements in the trace γ_2 only if they are over the locations not present in the trace γ_1. In fact, this kind of subsequence will be useful for analyzing efficient student solutions in Chapter 3.

We call this subsequence *full subsequence* and define it formally next.

Definition 2.11 (Full subsequence). *Let $\gamma_1 = (\ell_{1,1}, val_{1,1}) \cdots (\ell_{1,n}, val_{1,n})$, and $\gamma_2 = (\ell_{2,1}, val_{2,1}) \cdots (\ell_{2,m}, val_{2,m})$ be two computation traces over some set of locations Loc.*

Then, γ_1 is a full subsequence of γ_2, written $\gamma_1 \sqsubseteq_{full} \gamma_2$, when:

- *$\gamma_1 \sqsubseteq \gamma_2$, and additionally*
- *γ_1 and $\gamma_2|_{\{\ell_{1,1},\dots,\ell_{1,n}\}}$ have the same length.*

This definition ensures that there are the same number of trace elements at corresponding locations in both traces, while allowing extra locations in γ_2 that are not present in γ_1.

Example 2.12. *For example, given (γ_1 and γ_2 are the same as in the previous example):*

$$\begin{aligned}
\gamma_1 &= \qquad\qquad\quad (\ell_1, val_2) \qquad\qquad\qquad\qquad\quad \cdot (\ell_2, val_5) \cdot (\ell_1, val_6) \\
\gamma_2 &= (\ell_2, val_1) \cdot (\ell_1, val_2) \cdot (\ell_2, val_3) \cdot (\ell_3, val_4) \cdot (\ell_2, val_5) \cdot (\ell_1, val_6) \\
\gamma_3 &= \qquad\qquad\quad (\ell_1, val_2) \qquad\qquad\quad\; \cdot (\ell_3, val_4) \cdot (\ell_2, val_5) \cdot (\ell_1, val_6)
\end{aligned}$$

Then we have $\gamma_1 \sqsubseteq_{full} \gamma_3$ (and $\gamma_1 \sqsubseteq \gamma_3$), but $\gamma_1 \not\sqsubseteq_{full} \gamma_2$ (although $\gamma_1 \sqsubseteq \gamma_2$).

Comparison Relation

The subsequence operation considers two trace elements, or more precisely two values, equal when they are exactly the same. However, it might be interesting to consider two trace values *equivalent* even when they are not the same, but *related* in some way. In fact, this will be useful when analyzing two student solutions that differ only in the low-level data representation.

For this reason we extend the original subsequence relation by allowing arbitrary comparison functions, instead of the identity relation (as used in the original definition).

Definition 2.13 (Comparison subsequence). *Let $\gamma_1 = (\ell_{1,1}, val_{1,1}) \cdots (\ell_{1,n}, val_{1,n})$, and $\gamma_2 = (\ell_{2,1}, val_{2,1}) \cdots (\ell_{2,m}, val_{2,m})$ be two computation traces over some set of locations Loc.*

[4]The trace γ_2 also contains the additional location ℓ_3 with the element (ℓ_3, val_4).

Let $\delta : Loc \to 2^{Val \times Val}$ be a comparison function that maps program locations to equality relations on the value domain Val.

Then, γ_1 is an comparison subsequence of γ_2, written $\gamma_1 \sqsubseteq_\delta \gamma_2$, when:

- *There are indices $1 \leq k_1 < k_2 < \cdots k_n \leq m$, such that for all $1 \leq i \leq n$ we have $\ell_{1,i} = \ell_{2,k_i}$ and $(val_{1,i}, val_{2,k_i}) \in \delta(\ell_{1,i})$.*

We point out that with $\delta(\ell) = Id$ (where Id is the identity relation), for all $\ell \in Loc$, we obtain the original subsequence definition. Note that this definition allows us to relate different (although related by δ) elements of the trace.

We next discuss some examples of both the full and comparison subsequences.

Example 2.14. *Consider the following traces:*

$$\begin{aligned}
\gamma_1 &= \qquad\quad (\ell_1,1)\cdot(\ell_2,1) \qquad\qquad \cdot(\ell_2,2)\cdot(\ell_1,2) \qquad\qquad \cdot(\ell_2,3)\\
\gamma_2 &= (\ell_3,0)\cdot(\ell_4,1)\cdot(\ell_5,1)\cdot(\ell_3,3)\cdot(\ell_5,2)\cdot(\ell_4,2)\cdot(\ell_4,3)\cdot(\ell_5,3)\\
\gamma_3 &= (\ell_3,0)\cdot(\ell_4,1)\cdot(\ell_5,1)\cdot(\ell_3,3)\cdot(\ell_5,2)\cdot(\ell_4,2) \qquad\qquad \cdot(\ell_5,3)\\
\gamma_4 &= (\ell_3,1)\cdot(\ell_4,2)\cdot(\ell_5,2)\cdot(\ell_3,4)\cdot(\ell_5,3)\cdot(\ell_4,3) \qquad\qquad \cdot(\ell_5,4)
\end{aligned}$$

and let

$$\pi = \{\ell_1 \mapsto \ell_4, \ell_2 \mapsto \ell_5\} \text{ and } \delta_1(\ell_1) = \delta_2(\ell_2) = Id$$

be a mapping function and an (identity) comparison function, respectively.

In the previous section we established that $\gamma_1 \sqsubseteq^\pi \gamma_2$. However, because of the element $(\ell_4, 3)$ in γ_2 we have that $\gamma_1 \not\sqsubseteq^\pi_{full} \gamma_2$, that is, the trace γ_1 cannot be embedded in the trace γ_2, using the full subsequence.

The trace γ_3 is the same as the trace γ_2, but without the element $(\ell_4, 3)$, hence we have that $\gamma_1 \sqsubseteq^\pi_{full} \gamma_3$.

Finally, the trace γ_4 is the same as the trace γ_3, except that every value is incremented by one. Hence, by setting

$$\delta_2(\ell_1) = \delta_2(\ell_2) = \{(a, b) \mid a + 1 = b\}$$

that is, the comparison function δ_2 (on both ℓ_1 and ℓ_2) considers a and b as equal if $a + 1 = b$, then we have that $\gamma_1 \sqsubseteq^\pi_\delta \gamma_4$.

The Algorithm

We next comment on the algorithm for the two subsequence definitions. It is not difficult to see that the algorithm for the Embedding Problem (defined in Figure 2.2) works also with the $\sqsubseteq_{full}$ and $\sqsubseteq_\delta$ relations (instead of the original $\sqsubseteq$ relation). Further, it would also work for the combination of these two relations (full subsequence with comparison function). This is the case because the algorithm only assume that a relation is preserved when restricted to single locations, in order to construct a potential graph G.

Arbitrary trace relations Hence, the algorithm would work for any relation on the traces that is closed under location-projection; we define this formally next.

Definition 2.15 (Closure under location-projection)**.** *Let Loc be some set of program locations, and let $\mathcal{R} \subseteq \Gamma_{Loc} \times \Gamma_{Loc}$ be a binary relation over traces of Loc.*

$\mathcal{R}$ is closed under location-projection if: for all traces $\gamma_1, \gamma_2 \in \Gamma_{Loc}$, $\gamma_1 \mathcal{R} \gamma_2$ implies that, for all locations $\ell \in Loc$, $\gamma_1|_{\{\ell\}} \mathcal{R} \gamma_2|_{\{\ell\}}$.

By EMBED$_{\mathcal{R}}$ we denote the generalized version of the EMBED algorithm (Figure 2.2), that works on some relation (closed under location-projection) $\mathcal{R}$, instead of the original relation $\sqsubseteq$. The original algorithm is generalized at lines 5 and 11 to check whether the relation $\mathcal{R}$, instead of the relation $\sqsubseteq$, holds (resp. does not hold).

2.3.2 Traces

The traces (as defined in Definition 2.3) record a single value for each program location, and the trace embedding relation allows us to compare values at matching program locations. However, for our repair algorithm (in Chapter 4) we will be interested in a variation of this notion, where we want to compare *multiple values* at matching program locations, instead of only a single value. We motivate this with the following example.

```
x = 1
y = 2
return f(x, y)
```

(a) Program A

```
a = 2
b = 1
return f(b, a)
```

(b) Program B

Figure 2.3: Example of two simple programs.

Example 2.16. *Figure 2.3 shows two programs for which we would like to show that they have the same behavior (traces). Let us discuss some possible ways in which we could do this. Let us assume, for simplicity, that `f(1, 2) = 3`*

We could, as we did for the trace embedding problem, assign to each of the values (statements) a program location and assign the computed value to it. Then we would obtain the following traces:

$$\begin{aligned}\gamma_A &= (\ell_1, 1) \cdot (\ell_2, 2) \cdot (\ell_3, 3) \\ \gamma_B &= (\ell_1, 2) \cdot (\ell_2, 1) \cdot (\ell_3, 3)\end{aligned}$$

The problem is that in this way the traces capture the order in which the variables were assigned and they are different in the two programs.

We could record (in the traces) only the resulting (return) value and thus, in the both cases, obtain the single-element trace $(\ell_1, 3)$. However, we would also like to capture the values of the intermediate variables, that is, this way we loose important information.

Finally, we could record an assigned variable, together with a value assigned to it, in a way that abstracts over the order in which they were assigned. More precisely, we construct the following mappings for the two programs:

$$\{x \mapsto 1, y \mapsto 2, \mathit{return} \mapsto 3\}$$

$$\{a \mapsto 2, b \mapsto 1, \mathit{return} \mapsto 3\}$$

Hence, we change the way traces are obtained in the following way:

- A program location is assigned to a sequence of statements (i.e., to basic-blocks), instead of to each statement. See Section 4.2 for more details.
- For each program location we attach a mapping from program variables to computed values, instead of a single value.

In this way we abstract over the order in which program variables are assigned (on some program location), but still require that all matching variables in both traces are assigned the same values.

We now define this notion formally.

Definition 2.17 (Memory Trace)**.** *Let Var be a set of program variables. A memory $\sigma : \mathit{Var} \to \mathit{Val}$ is a mapping from program variables to computation values. Let Σ_{Var} denote the set of all memories over Var.*

A memory trace $\gamma \in (\Sigma_{\mathit{Var}})^*$ is a finite sequence of memories; let $\Gamma^m_{\mathit{Var}} = (\Sigma_{\mathit{Var}})^*$ denote the set of all memory traces over Var.

This trace definition does not directly fit the trace definition used in the previous sections (i.e., a sequence of location-value pairs), and hence we cannot apply the trace embedding algorithm directly to the memory traces. However, it is easy to transform memory traces to a representation analogous to the location-value pairs, and hence use the trace embedding algorithm on the transformed traces. We next define this formally.

Definition 2.18 (Variable trace)**.** *Let Var be a set of program variables, and let Val be a set of computation values.*

A variable trace $\gamma \in (\mathit{Var} \times \mathit{Val})^*$ is a finite sequence of variable-value pairs; let Γ_{Var} denote the set of all variable traces over Var.

Definition 2.19 (Variable trace construction)**.** *Let $\gamma = \sigma_1 \cdots \sigma_n \in \Gamma^m_{\mathit{Var}}$ be a memory trace over the variables $\mathit{Var} = \{v_1, \ldots, v_n\}$.*

We define a corresponding variable trace as a flattening of the original memory trace:

$$\mathcal{F}(\gamma) = (v_1, \sigma_1(v_1)) \cdots (v_n, \sigma_1(v_n)) \cdots (v_1, \sigma_n(v_1)) \cdots (v_n, \sigma_n(v_n))$$

Then we have $\mathcal{F}(\gamma) \in \Gamma_{\mathit{Var}}$.

We point out that all the development in the previous sections can be applied to the variable traces; the only difference is that here we use program variables *Var*, in place of program locations *Loc*. With the flattening function, all the development from the previous sections can be also applied to the memory traces. This is useful because in Chapter 4 we will base our technical development on the notion of memory traces.

In the following, we give explicit adaptations of our earlier definitions to memory traces.

Definition 2.20 (Variable mapping and rewriting)**.** *Let Var_1 and Var_2 be two sets of variables, let $\tau : Var_1 \to Var_2$ be an injective function, and let $\sigma \in \Sigma_{Var_1}$ be a memory over Var_1.*

Then we define $\tau(\sigma) = \{\tau(v) \mapsto \sigma(v) \mid v \in Var_1\}$, and lift it to the memory traces by setting $\tau(\sigma_1 \cdots \sigma_n) = \tau(\sigma_1) \cdots \tau(\sigma_n)$.

Definition 2.21 (Memory trace matching)**.** *Let Var_1 and Var_2 be two sets of variables, and let $\gamma_1 \in \Gamma^m_{Var_1}$ and $\gamma_2 \in \Gamma^m_{Var_2}$ be two memory traces over Var_1 and Var_2, respectively.*

If there exists a bijective function $\tau : Var_1 \to Var_2$, such that $\tau(\gamma_1) = \gamma_2$ we say that γ_1 and γ_2 match over τ, written $\gamma_1 \sim^\tau \gamma_2$. We say that τ is a matching witness.

We point out that this notion of matching requires that corresponding variables (related by τ) take the *same values* in the *same order* in both traces. Or put differently, the matching requires that the traces are equivalent, up to the renaming of variables by τ.

Finally, we define an equivalent matching notion for the variable traces, which allows us to use the Trace Embedding algorithm (defined in Figure 2.2 and generalized to arbitrary relations in Section 2.3.1) for the memory trace matching.

Definition 2.22 (Variable restriction and variable trace rewriting)**.** *Given some variable trace $\gamma \in \Gamma_{Var}$, and a set of variables $Var_2 \subseteq Var$, let $\gamma|_{Var_2}$ denote a sequence obtained from γ by deleting all pairs (v, val), where $v \notin Loc_2$.*

Let Var_1 and Var_2 be two sets of variables, let $\tau : Var_1 \to Var_2$ be an injective function, and let $\gamma = (v_1, val_1) \cdots (v_n, val_n)$ be a variable trace over Var_1. Then we define:

$$\tau(\gamma) = (\tau(v_1), val_1) \cdots (\tau(v_n), val_n)$$

Definition 2.23 (Variable trace matching)**.** *Let Var be a set of variables. We define a relation over variable traces, $\sim \subseteq (\Gamma_{Var} \times \Gamma_{Var})$, which is closed under the variable-projection as follows:*

$$\gamma_1 \sim \gamma_2 \textit{ iff } \gamma_1|_{\{v\}} = \gamma_2|_{\{v\}} \textit{ for all } v \in Var$$

Let $\gamma_1 \in \Gamma_{Var_1}$ and $\gamma_2 \in \Gamma_{Var_2}$ be two variable traces, over the variables Var_1 and Var_2, respectively. We say that γ_1 and γ_2 match, if there exists a bijective function $\tau : Var_1 \to Var_2$, such that $\tau(\gamma_1) \sim \gamma_2$, written $\gamma_1 \sim^\tau \gamma_2$.

Next, we formally state that the two notions of matching agree.

Proposition 2.24 (Equivalence of matching)**.** *Let Var_1 and Var_2 be two sets of variables, and let $\gamma_1 \in \Gamma^m_{Var_1}$ and $\gamma_2 \in \Gamma^m_{Var_2}$ be two memory traces over Var_1 and Var_2, respectively.*

Finally, let $\tau : Var_1 \to Var_2$ be a bijective function.

Then we have the following:

$$\gamma_1 \sim^\tau \gamma_2 \textit{ if and only if } \mathcal{F}(\gamma_1) \sim^\tau \mathcal{F}(\gamma_2)$$

Proof. (Sketch) First, we observe that, given any two memories σ_1 and σ_2 over the variables $Var = \{v_1, \ldots, v_n\}$, we have: $\sigma_1 = \sigma_2$ if and only if $\sigma_1(v_1) \cdots \sigma_1(v_n) = \sigma_2(v_1) \cdots \sigma_2(v_n)$.

Then the theorem follows by induction on the length of the traces γ_1 and γ_2 (they need to be of the same length for $\sim^\tau$ to hold). □

We conclude this section by illustrating the defined matching notions on an example.

Example 2.25. *Let $Var_1 = \{a, b, c\}$ and $Var_2 = \{x, y, z\}$ be two sets of variables.*

Next, consider the following two memory traces over the variables Var_1 and Var_2, respectively:

$$\begin{aligned}
\gamma_1 &= \{a \mapsto 3, b \mapsto 0, c \mapsto 0\} \cdot \{a \mapsto 3, b \mapsto 1, c \mapsto 3\} \cdot \{a \mapsto 3, b \mapsto 2, c \mapsto 5\} \\
&\quad \cdot \{a \mapsto 3, b \mapsto 3, c \mapsto 6\} \cdot \{a \mapsto 3, b \mapsto 3, c \mapsto 6\} \\
\gamma_2 &= \{x \mapsto 0, y \mapsto 0, z \mapsto 3\} \cdot \{x \mapsto 3, y \mapsto 1, z \mapsto 3\} \cdot \{x \mapsto 5, y \mapsto 2, z \mapsto 3\} \\
&\quad \cdot \{x \mapsto 6, y \mapsto 3, z \mapsto 3\} \cdot \{x \mapsto 6, y \mapsto 3, z \mapsto 3\}
\end{aligned}$$

The corresponding (flattened) variable traces are the following:

$$\begin{aligned}
\mathcal{F}(\gamma_1) &= (a,3) \cdot (b,0) \cdot (c,0) \cdot (a,3) \cdot (b,1) \cdot (c,3) \cdot (a,3) \cdot (b,2) \cdot (c,5) \\
&\quad \cdot (a,3) \cdot (b,3) \cdot (c,6) \cdot (a,3) \cdot (b,3) \cdot (c,6) \\
\mathcal{F}(\gamma_2) &= (x,0) \cdot (y,0) \cdot (z,3) \cdot (x,3) \cdot (y,1) \cdot (z,3) \cdot (x,5) \cdot (y,2) \cdot (z,3) \\
&\quad \cdot (x,6) \cdot (y,3) \cdot (z,3) \cdot (x,6) \cdot (y,3) \cdot (z,3)
\end{aligned}$$

With the bijective function $\tau : Var_1 \to Var_2$, defined with $\tau = \{a \mapsto z, b \mapsto y, c \mapsto x\}$ we have $\gamma_1 \sim^\tau \gamma_2$ and $\mathcal{F}(\gamma_1) \sim^\tau \mathcal{F}(\gamma_2)$.

2.4 Conclusion

We have presented a novel *dynamic relational program analysis*. The analysis is built upon the results of the manual code study and motivated by the need to distinguish different algorithmic strategies. The key observation behind the analysis is that the implementations of the same strategy compute the same *key-values* during execution on the same inputs.

We first define a basic version of the analysis, called the *Trace Embedding* problem, and then iteratively generalize different parts of the analysis. The program analysis presented in this chapter is, in its different instantiations, key building block for the feedback generation approaches presented in the next two chapters.

The main technical contributions of this chapter are the formal description of the analysis and the algorithm for the analysis.

CHAPTER 3

Performance Feedback

In this chapter we describe our approach for providing *performance* feedback on *correct* student solutions.

We start by discussing an overview of the approach on some examples in Section 3.1.

We then continue with the technical development of the approach: in Section 3.2 we describe some preliminaries required to discuss the core algorithms in Section 3.3, and some extensions to it in Section 3.4.

In Section 3.5 we describe the methodology how the approach could be used in practice, we discuss the implementation of the approach in Section 3.6, and in Section 3.7 we describe our experimental evaluation of the implementation and the obtained results.

3.1 Overview of the Approach

In this section we give an overview of our approach for providing performance feedback.

We first recapitulate the main observations behind our approach, discussed earlier in the thesis:

1. To provide performance feedback on a student's implementation (of some programming problem) we first need to determine the strategy used in the implementation (Section 1.4).
2. A strategy is specified by its *key-values* (for some inputs) and an implementation is determined to use this strategy if it has the same key values when executed on the same inputs (Section 2.1 and Section 2.2).

However, while discussing key-values, we left the following question unanswered:
How do we determine what are the key-values that define a strategy (on some input)?

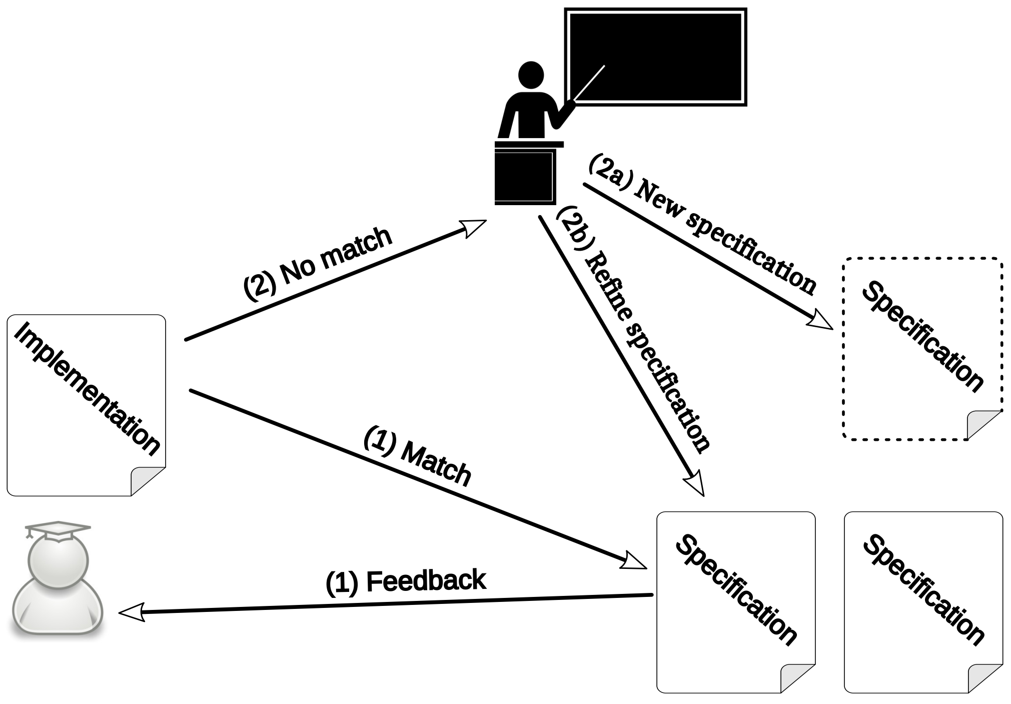


Figure 3.1: High-level overview of the methodology.

Specification and matching To that end, we propose a framework that allows a teacher to describe an algorithmic strategy by writing a *sample implementation* for each strategy. We refer to a sample implementation also as a *specification*. A teacher can annotate certain expressions (that produce key-values) in the specification using a special language statement **`observe`**.

The framework decides whether a student's implementation P_{impl} *matches* a teacher's specification P_{spec} by comparing their execution traces on common inputs, using the trace embedding notion (discussed in Section 2.2); as a reminder:

1. The execution trace of P_{spec}, generated by **`observe`** statements, is a subsequence of the execution trace of P_{impl}; and

2. For every *observed expression* in P_{spec} there is an expression in P_{impl} that generates the same values.

A high level description of our framework is given in Figure 3.1; parts (2), (2a) and (2b) can be ignored for now, they are explained below.

Next we discuss an example of a specification that uses the **`observe`** statement.

Example 3.1. *In Figure 1.6 (Section 1.4) we give the specification **CS**, for the implementations **C1** and **C2** (Figure 1.1 and Figure 1.4). The expressions that produce key-values (discussed in the previous example) are annotated with the **`observe`** statements:*

- *At lines 5 and 12 the **`observe`** statements annotate the iterated characters.*
- *At lines 10 and 17 the **`observe`** statements annotate the character count.*

It is not difficult to see that when executed on the input `s="aba"` and `t="baa"`, the annotated expressions produce the following sequence of values:

$$(a, b, a, 2, b, a, a, 2, a, b, a, 1, b, a, a, 1, a, b, a, 2, b, a, a, 2)$$

*Note that this is the same trace produced by the underlined expressions of **C1** and **C2** on the same input. That is, the trace of the specification is a subsequence of the traces of **C1** and **C2**.*

*Further, for each **`observe`** statement in the specification there is an expression in **C1** and **C2** that generates the same values. Hence, we say that implementations **C1** and **C2** match the specification.*

Methodology Next, we briefly describe how the specification mechanism is used to provide feedback to student solutions (this is described in more detail in Section 3.5). The high-level idea is also given in Figure 3.1.

The teacher is maintaining a list of specifications, and each specification has attached a textual feedback. When a student submits an implementation, one of two scenarios can happen:

1. The implementation matches some specification, and the student is presented with the feedback attached to the specification.
2. The implementation does not match any specification and the teacher is notified that there is an unmatched implementation.

In the (2.) scenario there are two sub-scenarios:

a. The student has written an implementation of a new algorithmic strategy for which there is no specification yet. In this case the teacher writes a new specification for this strategy.

b. The student has written an implementation of an algorithmic strategy for which there is a specification, but the implementation does not match this specification. In this case the teacher *refines* the existing specification to also match the new implementation.

```
void Puzzle(string s, string t) {
  if (nd1)
    s = s.ToUpperInvariant();
  char[] ca = s.ToCharArray();
  Array.Sort(ca);
  if (nd2) Array.Reverse(ca);
  observe (ca);
}
```

Figure 3.2: A specification for the sorting strategy using non-deterministic variables (**SS'**).

Example 3.2. *Imagine that the a teacher has already written the specification **CS** in Figure 1.6 (discussed above).*

*When a student submits the implementations **C1** or **C2**, the implementation is going to match the specification and the students are going to be given the appropriate feedback; as mentioned in Section 1.4, this might be: "Calculate the number of characters in each string in a preprocessing phase, instead of each iteration of the main loop". This is scenario (1.) from the above.*

*When a student submits an implementation of any other strategy, for example **S2** (discussed in Section 1.4, and given in Figure 1.5, the teacher will be notified of the new implementation, examine it, and determine that there is a need for a new specification (for the sorting strategy). This is scenario (2a.) from the above.*

*When a student submits the implementation **C3** (discussed in Section 1.4, and given in Figure 1.4), the teacher will be notified of the new implementation, examine it, and determine that the counting specification needs to be refined to match **C3**. This is scenario (2b.) from the above.*

We are not going to describe the refinement process in more detail here here (see Section 3.5 for a detailed discussion). However, we are going to discuss one important refinement mechanism available to the teacher.

Non-deterministic choice Different implementations of the same strategy can have minor differences in key-values. Hence, the implementations will produce different, although related, traces on the same inputs. Without an additional help a teacher would need to write multiple specifications for the same strategy. To address variations in implementation details, the framework allows a teacher to use *non-deterministic* Boolean variables in the specification language. However, the non-determinism is fixed before the execution, and thus such a choice is merely a syntactic sugar to succinctly represent *multiple similar specifications.* A specification with n different non-deterministic variables corresponds to 2^n different specifications.

We discuss this on an example.

Example 3.3. *In Figure 1.6 (b) (Section 1.4) we give the specification **SS**, for the* sorting strategy*. This specification observes the first input sorted. However, we have also seen examples where student' implementations additionally convert the input strings to the upper case and sort the strings backwards.*

To specify these variations the teacher would need to write 4 different specifications for the sorting strategy:

- *Original (lower case) and sorted in the original direction;*
- *Original (lower case) and sorted backwards;*
- *Upper case and sorted in the original direction; and*
- *Upper case and sorted backwards.*

*By using non-deterministic variables the teacher can represent that succinctly, as shown in the specification **SS'** in Figure 3.2. The teacher uses the following non-deterministic variables:*

- $\texttt{nd}_1$ *at line 2 to convert the first input string to the upper case; and*
- $\texttt{nd}_2$ *at line 6 to reverse the sorted first input.*

For the input string s="aba", depending on the values of the non-deterministic variables, the teacher observes (at line 7) the following:

$\texttt{nd}_1$	$\texttt{nd}_2$	*The observed value*
false	*false*	*aab*
false	*true*	*AAB*
true	*false*	*baa*
true	*true*	*BAA*

In Section 3.3 we discuss how we match an implementation to a specification with non-deterministic variables.

3.2 Program Model

In this section we formally present an imperative language that supports standard constructs for writing implementations, and additional constructs for writing specifications. We then define how are execution traces obtained from implementations and specifications.

$$\begin{array}{lcl}\text{Expressions } e & ::= & d \mid v \mid v_1\ op_{bin}\ v_2 \mid op_{un}\ v \mid v_1[v_2] \\ \text{Statements } s & ::= & v := e \mid v_1[v_2] := e \mid v := \mathbf{f}(v_1, \ldots, v_n) \mid s_1; s_2 \mid \texttt{skip} \\ & \mid & \texttt{while}\ v\ \texttt{do}\ s \mid \texttt{if}\ v\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \\ & \mid & \texttt{observe}\,(v) \mid \texttt{observeFun}\,(\mathbf{f}(v_1, \ldots, v_n))\end{array}$$

Figure 3.3: The syntax for implementations and specifications.

3.2.1 The Language

Definition 3.4 (Data domain and library functions)**.** *A data value $d \in D$ is a value from some data domain set D. D also contains a special value $*$ that represents any data value, and comes equipped with some equality relation $=_D \subseteq D \times D$.*

Let F be a (finite) set of library functions.

The data domain D, the corresponding equality relation $=_D$, and the set of library functions F, are defined by a concrete language. For example, for C#, D contains all integers, characters, arrays, hash-sets, ..., and $(d_1, d_2) \in =_D$ iff d_1 and d_2 are of the same type, and comparison by the `Equals` method returns `true`. Similarly, for C#, F contains the standard set of library functions.

Definition 3.5 (Expressions)**.** *Let variable $v \in Var$ belong to a (finite) set of variables Var.*

The set of expressions E is defined in BNF notation in Figure 3.3, where op_{bin} ranges over a set of binary operators (e.g., $+$, $\cdot$, $\leq$), and op_{un} ranges over a set of unary operators (e.g., $-$, $\neg$).

An expression is a either a data value d, a variable v, a binary operator applied to variables v_1 and v_2, a unary operator applied to a variable v, or an array access $v_1[v_2]$.

Definition 3.6 (Value domain, equality relation)**.** *Given a data domain D, and a set of library functions F, we define $Val = D \cup (F \times D^*)$ as the value domain.*

Further, let $\mathcal{E} = 2^{Val \times Val}$ be the set of all (binary) relations over Val. The default equality relation $\xi_{def} \in \mathcal{E}$ is defined as follows:

- *$(x, y) \in \xi_{def}$ if and only if $x = *$ or $y = *$, or $(x, y) \in =_D$.*
- *$(\mathbf{f}(x_1, \ldots, x_n), \mathbf{g}(y, \ldots, y_n)) \in \xi_{def}$ if and only if $f = g$ and $(x_i, y_i) \in \xi_{def}$, for all $1 \leq i \leq n$.*

The definition of the (default) equality relation is important for trace comparison, i.e., for deciding whether two trace elements are equal. Later (see Section 3.4) we also allow user-specified equality relations.

Definition 3.7 (Statements)**.** *The syntax of statements is defined in BNF notation in Figure 3.3.*

Let Loc be a set of program locations, then each statement s in a program is assigned a unique location *$\ell \in Loc$, denoted with $\ell : s$.*

The statements of the language are the standard ones, and consist of: assignments to variables and array elements, `skip` statement, looping and branching constructs. There are also two special *observe* statements, which are available only to the teacher, and not to the student. We discuss the observe statements in more detail in Section 3.2.3 below.

Note that the syntax of the language ensures that the programs are in the *three-address code*. That is, operators can only be applied to variables, but not to arbitrary expressions. The three address code enables observing any expression in the program by observing only variables. However, any expression can be *automatically* translated into three address code by assigning each sub-expression to a new variable, and this is done automatically in the framework implementation, that is, teachers and students write normal C# code.

Example 3.8. *If we have an assignment $v_1 := v_2 + \mathbf{f}(v_3 + v_4 * v_5, v_6)$, it can be automatically transformed into a sequence of assignments in three-address code:*

- $v_7 := v_4 * v_5$,
- $v_8 := v_3 + v_7$,
- $v_9 := \mathbf{f}(v_8, v_6)$, *and*
- $v_1 := v_2 + v_9$.

Semantics We assume a standard imperative semantics for programs written in the given language. For example, for C#, we assume the standard semantics of C#. The two observe statements have the same semantic meaning as the `skip` statement.

Definition 3.9 (Computation trace)**.** *A computation trace γ over some set of locations Loc is a finite sequence of location-value pairs $(Loc \times Val)^*$. Let Γ_{Loc} be the set of all traces over Loc.*

Given some trace $\gamma \in \Gamma_{Loc}$, and some set of locations $Loc_2 \subseteq Loc$, let $\gamma|_{Loc_2}$ denote the sequence obtained from γ by deleting all pairs (ℓ, val), where $\ell \notin Loc_2$.

Example 3.10. *If we have*

$$\begin{array}{rcl} Loc & = & \{\ell_1, \ell_2, \ell_3, \ell_4\} \\ Loc_2 & = & \{\ell_2, \ell_3\} \\ \gamma & = & (\ell_1, val_1), (\ell_3, val_2), (\ell_3, val_3), (\ell_1, val_4), (\ell_2, val_5), (\ell_3, val_6) \\ \textit{then} & & \\ \gamma|_{Loc_2} & = & (\ell_3, val_2), (\ell_3, val_3), (\ell_2, val_5), (\ell_3, val_6) \end{array}$$

Functions We defined our program model without functions for ease of exposition. However, this represents no restriction and it is straightforward to extend to programs with functions (also recursive). However, we point out that in case of non-recursive function we treat them as if they were inlined. This has the benefit that locations inside a function are considered different based on the location where the function was called from; in other words we get context-sensitivity inside functions. We illustrate this on an example.

Example 3.11. *Consider the implementation **C1** (from Figure 1.1, discussed in Section 1.1).*

Without context-sensitivity the underlined expression at line 15 would be considered the same location, when the function `countChars` is called from line 5 or 6. However, we want to distinguish whether the expression at line 15 is computed by calling `countChars` from line 5 or from line 6.

Definition 3.12 (Memory)**.** *Given some set of variables Var, we define the memory $\sigma : Var \to D$ to be a mapping from program variables to data values. Let Σ_{Var} denote all memories over variables Var.*

We illustrate the defined program model and the semantics below, after we define how a trace is constructed for implementations and specifications.

3.2.2 Student Implementation

Now we describe how a computation trace γ is constructed for a student implementation P_{impl}, during execution on some input σ_{input}. Note that a student cannot use any of the observe statements.

Before P_{impl} is executed, the computation trace is initialized to the empty sequence $\gamma = \epsilon$. Then P_{impl} is executed on the input σ_{input}, according to the semantics of the language. During the execution, we append location-value pairs to γ *after* the following statements are executed:

- For an assignments. $\ell : v_1 := e$ or $\ell : v_1[v_2] := e$, we append $(\ell, \sigma(v_1))$ to γ.
- For a library function call, $\ell : v := \mathbf{f}(v_1, \ldots, v_n)$, we append $(\ell, (\mathbf{f}, \sigma(v), \sigma(v_1), \ldots, \sigma(v_n)))$ to γ.

Here, $\sigma(v)$ denotes a current value of a variable v. Note that for the assignment to some array v_1, we append the complete array $\sigma(v_1)$ to the trace. The trace constructed by the described process is denoted by $\gamma = [\![P_{impl}]\!](\sigma_{input})$.

Next we show on an example how a trace is obtained from a student implementation.

```
 1 bool Puzzle(s, t) {
 2   i = 0;
 3   n = |s|;
 4   while (i < n) {
 5     c = s[i];
 6     ss = Split(s, c);
 7     cnt1 = |ss|;
 8     st = Split(t, c);
 9     cnt2 = |st|;
10     i = i + 1;
11   }
12 }
```

Figure 3.4: Representation (simplified) of the program **C3** in our program model.

Example 3.13. *Figure 3.4 shows a simplified (after conversion to three-address-code and code slicing for simplicity) model of the implementation* ***C3*** *(from Figure 1.4 in Section 1.4).*

Note that every assignment is on its own line; therefore, we denote line i by location $\ell_{\textit{\textbf{C3}},i}$.

If we run the program on `s="aab"` *and* `t="aba"` *we obtain the following trace:*

$$\begin{aligned}\gamma_{\textit{\textbf{C3}}} \quad = \quad & (\ell_{\textit{\textbf{C3}},2}, 0) \cdot (\ell_{\textit{\textbf{C3}},3}, 3) \cdot (\ell_{\textit{\textbf{C3}},5}, a) \cdot (\ell_{\textit{\textbf{C3}},6}, (\texttt{Split}, [, , b], aab, a)) \cdot (\ell_{\textit{\textbf{C3}},7}, 3) \\ & \cdot (\ell_{\textit{\textbf{C3}},8}, (\texttt{Split}, [, b,], aba, a)) \cdot (\ell_{\textit{\textbf{C3}},9}, 3) \cdot (\ell_{\textit{\textbf{C3}},10}, 1) \cdot (\ell_{\textit{\textbf{C3}},5}, a) \cdots\end{aligned}$$

3.2.3 Teacher Specification

Now we describe how a computation trace γ is constructed for the specification of the teacher P_{spec}, during execution on some input σ_{input}. The teacher uses `observe` $(\cdot)$ and `observeFun` $(\cdot)$ statements to specify the key values she wants to observe during the execution of the specification P_{spec}.

Before P_{spec} is executed, the computation trace is initialized to the empty sequence $\gamma = \epsilon$. Then P_{spec} is executed on the input σ_{input}, according to the semantics of the language. During the execution, we append location-value pairs to trace γ *only when the observe statements* are executed:

- For a regular observe statement ℓ : `observe` (v), we append $(\ell, \sigma(v))$ to γ.
- For a library function observe statement ℓ : `observeFun` $(\mathbf{f}(v_1, \ldots, v_n))$, we append $(\ell, (\mathbf{f}, \sigma(v_1), \ldots, \sigma(v_n)))$ to γ.

The trace constructed by the described process is denoted by $\gamma = [\![P_{spec}]\!](\sigma_{input})$.

Non-deterministic variables We have already motivated non-deterministic variables in Section 3.1: they are useful for specifying variations of different implementations of the same strategy.

```
 1 void Puzzle(string s, string t) {
 2   for (int i = 0; i < s.Length; ++i) {
 3     int cnt1 = 0, cnt2 = 0;
 4     observe (s[i]);
 5     for (int j = 0; j < s.Length; ++j) {
 6       if (nd1) observe (s[j]);
 7       if (s[j] == s[i]) {
 8         cnt1++;
 9       }
10     }
11     if (!nd1) observeFun (Split(*, *, *));
12     observe (nd1 ? cnt1 : cnt1 + 1);
13     for (int j = 0; j < t.Length; ++j) {
14       if (nd1) observe (t[j]);
15       if (t[j] == s[i]) {
16         cnt2++;
17       }
18     }
19     if (!nd1) observeFun (Split(*, *, *));
20     observe (nd1 ? cnt2 : cnt2 + 1);
21   }
22 }
```

```
 1 Puzzle(s, t) {
 2   i = 0;
 3   n = |s|;
 4   cnt1 = cnt2 = 0;
 5   while (i < n) {
 6     c = s[i];
 7     observe (c);
 8     j = 0;
 9     m = |s|;
10     while (j < m) {
11       c2 = s[j];
12       if (nd1) observe (c2);
13       if (c1 == c2) cnt1++;
14       j = j + 1;
15     }
16     if (!nd1)
17       observeFun (Split(*, *, *));
18     observe (nd1 ? cnt1 : cnt1+1);
19     j = 0;
20     m = |t|;
21     while (j < m) {
22       c2 = t[j];
23       if (nd1) observe (c2);
24       if (c1 == c2) cnt2++;
25       j = j + 1;
26     }
27     if (!nd1)
28       observeFun (Split(*, *, *));
29     observe (nd1 ? cnt2 : cnt2+1);
30     i = i + 1;
31   }
32 }
```

Figure 3.5: The specification **CS'** and its (simplified) representation in our program model.

The teacher can use in specifications some finite set of *Boolean non-deterministic variables* $B = \{nd_1, \ldots, nd_n\} \subseteq Var$. Non-deterministic variables are similar to the input variables, in the sense that are assigned before the program is executed (we discuss this further in Section 3.3 when we discuss the matching algorithm). Note that this results in 2^n different program behaviors for a given input.

Next we discuss on an example how a trace is obtained from a teacher specification with a non-deterministic variable.

Example 3.14. *Figure 3.5 shows the specification **CS'** (for the counting strategy) and its simplified (after conversion to three-address-code and code slicing for simplicity) representation in our model. This specification uses the constructs **`observe`** and **`observeFun`**, and the non-deterministic variable* $\texttt{nd}_1$.

*Same as in the previous example, every assignment and **`observe`** statement is on its own line; therefore, we denote line i by location $\ell_{\mathbf{CS'},i}$.*

If we run the program on `s="aab"` *and* `t="aba"` *we obtain the following two traces, depending on the choice for the non-deterministic variable* $\mathtt{nd}_1$*:*

$$\begin{aligned}
\gamma_{\textit{CS'},\textit{true}} \quad = \quad & (\ell_{\textit{CS'},7}, a)\cdot \\
& (\ell_{\textit{CS'},12}, a)\cdot(\ell_{\textit{CS'},12}, a)\cdot(\ell_{\textit{CS'},12}, b)\cdot(\ell_{\textit{CS'},18}, 2)\cdot \\
& (\ell_{\textit{CS'},23}, a)\cdot(\ell_{\textit{CS'},23}, b)\cdot(\ell_{\textit{CS'},23}, a)\cdot(\ell_{\textit{CS'},29}, 2)\cdot \\
& (\ell_{\textit{CS'},7}, a)\cdot \\
& (\ell_{\textit{CS'},12}, a)\cdot(\ell_{\textit{CS'},12}, a)\cdot(\ell_{\textit{CS'},12}, b)\cdot(\ell_{\textit{CS'},18}, 2)\cdot \\
& (\ell_{\textit{CS'},23}, a)\cdot(\ell_{\textit{CS'},23}, b)\cdot(\ell_{\textit{CS'},23}, a)\cdot(\ell_{\textit{CS'},29}, 2)\cdot \\
& (\ell_{\textit{CS'},7}, b)\cdot \\
& (\ell_{\textit{CS'},12}, a)\cdot(\ell_{\textit{CS'},12}, a)\cdot(\ell_{\textit{CS'},12}, b)\cdot(\ell_{\textit{CS'},18}, 1)\cdot \\
& (\ell_{\textit{CS'},23}, a)\cdot(\ell_{\textit{CS'},23}, b)\cdot(\ell_{\textit{CS'},23}, a)\cdot(\ell_{\textit{CS'},29}, 1)
\end{aligned}$$

$$\begin{aligned}
\gamma_{\textit{CS'},\textit{false}} \quad = \quad & (\ell_{\textit{CS'},7}, a)\cdot \\
& (\ell_{\textit{CS'},17}, (\texttt{Split}, *, *, *))\cdot(\ell_{\textit{CS'},18}, 3)\cdot(\ell_{\textit{CS'},28}, (\texttt{Split}, *, *, *))\cdot(\ell_{\textit{CS'},29}, 3) \\
& (\ell_{\textit{CS'},7}, a)\cdot \\
& (\ell_{\textit{CS'},17}, (\texttt{Split}, *, *, *))\cdot(\ell_{\textit{CS'},18}, 3)\cdot(\ell_{\textit{CS'},28}, (\texttt{Split}, *, *, *))\cdot(\ell_{\textit{CS'},29}, 3) \\
& (\ell_{\textit{CS'},7}, b)\cdot \\
& (\ell_{\textit{CS'},17}, (\texttt{Split}, *, *, *))\cdot(\ell_{\textit{CS'},18}, 2)\cdot(\ell_{\textit{CS'},28}, (\texttt{Split}, *, *, *))\cdot(\ell_{\textit{CS'},29}, 2)
\end{aligned}$$

3.3 Algorithms

In this section we formally discuss the notion of *matching* between an implementation and a specification, and the corresponding algorithms. The notion of matching builds on top of the *Trace Embedding*, defined and discussed in Chapter 2.

Partial and full matching We distinguish two types of matching: *partial* and *full*. The partial marching is used to define *inefficient specifications* and to match *inefficient implementations*, while the full matching is used to define *efficient specifications* and to match *efficient implementations*. A teacher has to specify which of the two matching criterion is to be used in a specific case, that is, if a specification is efficient or inefficient.

Intuitively, when an implementation matches an inefficient specification (using partial matching), this means that the implementation is at least as inefficient as the specification. When an implementation matches an efficient specification (using full matching), this means that the implementation is as efficient as the specification. This is further discussed later in the section when we discuss each of the matching types.

One might wonder why do we need efficient specifications at all; that is, if the student already wrote an efficient implementation, there is no need for feedback. However, we need to distinguish the cases when there is no feedback because an implementation did not match any specification (we remind the reader about the discussion about the methodology in Section 3.1), and when the student wrote an efficient implementation. We discuss this further in Section 3.5, when we discuss the methodology in more detail.

```
1  def MATCHES(Specification $P_{spec}$, Implementation $P_{impl}$, Inputs $I$):
2      $Loc_1$ = observed locations in $P_{spec}$
3      $Loc_2$ = assignment locations of $P_{impl}$
4      for all $\sigma \in I$:
5          $\gamma_{2,\sigma}$ = $[\![P_{impl}]\!](\sigma)$
6      $B_{P_{spec}}$ = non-deterministic variables in $P_{spec}$
7      for all assignments $\sigma_{nd}$ to $B_{P_{spec}}$:
8          for all $\sigma \in I$:
9              $\gamma_{1,\sigma}$ = $[\![P_{spec}]\!](\sigma \cup \sigma_{nd})$
10         if EMBED$_{\sqsubseteq}$(($\gamma_{1,\sigma})_{\sigma \in I}$, $(\gamma_{2,\sigma})_{\sigma \in I}$, $Loc_1$, $Loc_2$):
11             return True
12     return False
```

Figure 3.6: Algorithm for the matching problem.

3.3.1 Partial Matching

We now define the notion of *partial matching* (also referred to simply as *matching*), which is used to check whether an implementation involves (at least) those inefficiency issues that underlie a given inefficient specification.

Definition 3.15 (Partial Matching)**.** *Let P_{spec} be a specification with observed locations Loc_1, and let P_{impl} be an implementation whose assignment statements are labeled by Loc_2.*

Then, implementation P_{impl} (partially) matches specification P_{spec}, on a set of inputs I, if and only if there exists a mapping function $\pi : Loc_1 \to Loc_2$, and an assignment to non-deterministic variables σ_{nd}, such that $\gamma_{1,\sigma} \sqsubseteq^{\pi} \gamma_{2,\sigma}$, for all input valuations $\sigma \in I$, where $\gamma_{1,\sigma} = [\![P_{spec}]\!](\sigma \cup \sigma_{nd})$, $\gamma_{2,\sigma} = [\![P_{impl}]\!](\sigma)$.

We recall that we defined the $\sqsubseteq$ relation in Definition 2.6 (Section 2.2).

Figure 3.6 shows our algorithm, MATCHES, for testing if an implementation P_{impl} (partially) matches a given specification P_{spec} over a given set of inputs valuations I. In lines 4-5, the implementation P_{impl} is executed on all input valuations $\sigma \in I$. In line 7, the algorithm iterates through all assignments σ_{nd} to the non-deterministic variables $B_{P_{spec}}$ of the specification P_{spec}. In lines 8-9, the specification P_{spec} is executed on all input valuations $\sigma \in I$. With both sets of traces, $(\gamma_{1,\sigma})_{\sigma \in I}$ and $(\gamma_{2,\sigma})_{\sigma \in I}$, available, line 10 calls the algorithm EMBED$_{\sqsubseteq}$ which returns `True` if there exists a trace embedding.

We remind the reader that we have given the EMBED [1] algorithm in Figure 2.2 and discussed it in Section 2.2 and Section 2.3. Next, we discuss the MATCHES algorithm on an example.

Example 3.16. *In Section 3.2 we have discussed the implementation **C3** (Example 3.13) and the specification **CS'** (Example 3.14).*

[1]We omit the subscript $\sqsubseteq$ when it is clear from the context.

In those examples we have given the traces for the implementation and the specification when executed on the input `s="aab"` and `t="aba"`. We point out that the matching algorithm works on a set of the inputs, but to make the presentation easier to follow we discuss the algorithm on this single input. This results in a single trace, instead of a set of traces.

We obtained the trace $\gamma_{\boldsymbol{C3}}$ from the implementation, and the traces $\gamma_{\boldsymbol{CS'},true}$ and $\gamma_{\boldsymbol{CS'},false}$ from the specification, depending on the value of the non-deterministic variable $\texttt{nd}_1$.

In this case the algorithm first calls the algorithm Embed *with the traces $\gamma_{\boldsymbol{C3}}$ and $\gamma_{\boldsymbol{CS'},true}$. There is no embedding between those two traces, hence* Embed *returns `False`, and the algorithm continues execution.*

Next, the algorithm calls Embed *with the traces $\gamma_{\boldsymbol{C3}}$ and $\gamma_{\boldsymbol{CS'},false}$. There is an embedding between these two traces, hence* Embed *returns `True`, and therefore the matching algorithm also returns `True`.*

3.3.2 Full Matching

Next we define the notion of *full matching*, which is used to match implementations against efficient specifications. To ensure that an implementation has *at most* the complexity of the efficient implementation, we put some additional constraints on the matching: we require that for *every loop* and *every library function call* in the implementation there is a corresponding loop and library function call in the specification. We start by defining these notions formally.

Definition 3.17 (Loop Iterations)**.** *We extend the construction of the implementation trace (defined in Section 3.2.2): for each statement ℓ : `while` v `do` s, we additionally append element $(\ell, \bot)$ to the trace whenever the loop body s is entered. We call $(\ell, \bot)$ a loop iteration.*

Definition 3.18 (Observed Loop Iterations and Library Function Calls)**.** *We extend the relation $\sqsubseteq_{full}$ defined in Definition 2.11.*

Let $\gamma_1 = (\ell_{1,1}, val_{1,1}) \cdots (\ell_{1,n}, val_{1,n})$, and $\gamma_2 = (\ell_{2,1}, val_{2,1}) \cdots (\ell_{2,m}, val_{2,m})$ be two traces (over some set of locations Loc). Next, let $\gamma_1 \sqsubseteq_{full} \gamma_2$; hence there is an injective function $\rho : \{1, \ldots, n\} \to \{1, \ldots, m\}$ that maps matched elements of γ_1 to the elements of γ_2.

We say that all loops are observed (in γ_2) if and only if: for each pair $(\ell_{2,i_1}, \bot) \in \gamma_2$ and $(\ell_{2,i_2}, \bot) \in \gamma_2$ where $i_1 < i_2$ and $\ell_{2,i_1} = \ell_{2,i_2}$, there is $1 \le j \le n$, such that $i_1 < \rho(j) < i_2$.

We say that all library calls are observed (in γ_2) if and only if: for each $(\ell_{2,i}, \mathtt{f}(val_1, \ldots, val_k)) \in \gamma_2$, there is $1 \le j \le n$, such that $\rho(j) = i$.

We set $\gamma_1 \sqsubseteq_{fullObserve} \gamma_2$ if and only if:

- *$\gamma_1 \sqsubseteq_{full} \gamma_2$, and*
- *all loops and library calls are observed in γ_2 (as defined above).*

In other words, we require that between any two iterations of the same loop in the implementation there exists some observed location in the specification, and that each function call in the implementation is also observed in the specification.

We point out that $\sqsubseteq_{\mathit{fullObserve}}$ is *closed under location-projection* (as defined in Definition 2.15), and hence we can use the $\text{EMBED}_{\sqsubseteq_{\mathit{fullObserve}}}$ algorithm.

Definition 3.19 (Full Matching)**.** *Let P_{spec} be a specification with observed locations Loc_1, and let P_{impl} be an implementation whose assignment statements are labeled by Loc_2.*

Then, implementation P_{impl} fully matches specification P_{spec}, on a set of inputs I, if and only if there exists a mapping function $\pi : Loc_1 \to Loc_2$, and an assignment to non-deterministic variables σ_{nd}, such that $\gamma_{1,\sigma} \sqsubseteq^{\pi}_{\mathit{fullObserve}} \gamma_{2,\sigma}$, for all input valuations $\sigma \in I$, where $\gamma_{1,\sigma} = [\![P_{spec}]\!](\sigma \cup \sigma_{nd})$, $\gamma_{2,\sigma} = [\![P_{impl}]\!](\sigma)$.

However, it might be quite tedious for a teacher to *exactly specify* all possible loop iterations and all library function calls that might be used in different efficient implementations. Therefore, we equip the defined language with two additional constructs to simplify the specification task.

Definition 3.20 (Cover Statements)**.** *We extend the language defined in Figure 3.3 with two additional cover statements: ℓ : `cover` (v) and ℓ : `cover` $(\mathbf{f}(v_1, \ldots, v_n))$ (that are available only in the specifications).*

The first statement allows the $\sqsubseteq_{\mathit{fullObserve}}$ relation to relate ℓ (in the specification) to a location in the implementation that appears at most $\sigma(v)$ times for each appearance of ℓ, where $\sigma(v)$ is the current value of the variable v.

The second statement is the same as the statement ℓ : `observeFun` $(\mathbf{f}(v_1, \ldots, v_n))$, except that we allow the $\sqsubseteq_{\mathit{fullObserve}}$ relation to not relate ℓ to any location in the implementation.

This enables the teacher to:

1. *Cover any loop* with up to $\sigma(v)$ iterations, with the first cover statement, and
2. Specify that the function $\mathbf{f}(v_1, \ldots, v_n)$ *may or may not appear* in the implementation, with the second cover statement.

We point out that it is easy to extend the $\sqsubseteq_{\mathit{fullObserve}}$ relation to respect the definition of the cover statements. Next, we discussed the defined notions on an example.

Example 3.21. *We already mentioned an efficient implementation **E1** for the Anagram problem in Figure 1.3 (c). Figure 3.7 (a) shows another efficient implementation **E2** for the Anagram problem. The implementation **E2** is similar to the implementation **E1**, but instead of using two arrays to count the character occurrences for each of the input*

```
 1 bool Puzzle(string s, string t) {
 2   if (s.Length != t.Length)
 3     return false;
 4
 5   char[] cs = s.ToCharArray();
 6   char[] ct = t.ToCharArray();
 7
 8   int[] hash = new int[256];
 9
10   for (int i=0; i<255; ++i) {
11     hash[i] = 0;
12   }
13   foreach (char ch in cs) {
14     hash[(int)ch]++;
15   }
16   foreach (char ch in ct) {
17     hash[(int)ch]--;
18   }
19   for (int i=0; i<255; ++i) {
20     if (hash[i] < 0)
21       return false;
22   }
23   return true;
24 }
```

(a) Efficient / Difference (**E2**).

```
 1 void Puzzle(string s, string t) {
 2   if (nd1){
 3     string tt = t;
 4     t = s;
 5     s = tt;
 6   }
 7   int[] cs = new int[256];
 8   int ct = new int[256];
 9
10   cover (ToCharArray());
11   cover (ToCharArray());
12   cover (255);
13
14   for (int i = 0; i < s.Length; ++i) {
15     cs[(int)s[i]]++;
16     observe (cs);
17   }
18   for (int i = 0; i < t.Length; ++i) {
19     if (nd2) {
20       cs[(int)t[i]]--;
21       observe (cs);
22     } else {
23       ct[(int)t[i]]++;
24       observe (ct);
25     }
26   }
27   cover (255);
28 }
```

(b) Efficient specification (**ES**).

Figure 3.7: Examples for the efficient implementation and specification.

strings, it uses one array to count the character occurrences of the first input string, and subtracts the count of the character occurrences of the second input string.

*Figure 3.7 (b) shows a specification **ES** for an efficient strategy, inspired by the implementations **E1** and **E2**, for the Anagram problem.*

The teacher observes computed values (arrays with the count of character occurrences) at lines 16, 21 and 24. Further the teacher uses two non-deterministic variables:

- `nd`$_1$ *at line 2 to swap the input strings, and*
- `nd`$_2$ *at line 19 to choose whether implementations count the number of characters in each string (inspired by **E1**), or substract one number from another (inspired by **E2**).*

*Finally, the teacher uses **`cover`** statements to:*

- *At lines 10 and 11 to allow for up to two `ToCharArray` library function calls, and*
- *At lines 12 and 27 to allow for up to two loops with at most 255 iterations.*

```
1  bool Puzzle(string s, string t) {
2    if(s.Length != t.Length)
3      return false;
4    Char[] taux = t.ToCharArray();
5    for(int i = 0; i < s.Length; i++) {
6      Char sc = s[i];
7      Boolean exists = false;
8      for(int j = 0; j < t.Length; j++) {
9        if(sc == taux[j]) {
10         exists = true;
11         taux[j] = '-';
12         break;
13       }
14     }
15     if(exists == false)
16       return false;
17   }
18   return true;
19 }
```

Figure 3.8: Removing / Manual 2 (**R4**)

3.4 Extensions

In this section, we discuss useful extensions to the core material presented so far. These extensions are part of our implementation, but we discuss them separately to make the presentation easier to follow.

3.4.1 Custom Data Equality

Another source of variations in implementations of the same strategy is data-representation.

That is, implementations might implement the same strategy using different representation of the data, and therefore key-values will be different, although related or similar.

We discuss this on an example.

Example 3.22. *In Figure 1.5 (Section 1.4) we have given the implementation **R3**; this implementation implements the removing strategy (also discussed in Section 1.4). Figure 3.8 shows the implementation **R4**, also implementing the removing strategy.*

*The implementations **R3** and **R4** implement the removing strategy in almost identical ways — the only difference is at lines 11 in both programs: the former implementation marks a character removed from a string with `'#'`, the latter with `'-'`. Thus, the only difference is in the data representation of the removed characters.*

The difference in the discussed example could be handled using non-deterministic variables, however we would need to add a new such variable for each new character used to mark removed characters.

Instead we extend the language to allow the teacher to specify *custom data equality* over the observed values.

```
void Puzzle(string s, string t) {
  if (nd₁) {
    string tt = t;
    t = s;
    s = tt;
  }
  for (int i = 0; i < s.Length; ++i) {
    if (s.Substring(i) == t) return;
    int ni = nd₂ ? i : s.Length - i - 1;
    int k = nd₃ ? t.IndexOf(s[ni])
                : t.LastIndexOf(s[ni]);
    t = t.Remove(k, 1);
    observe (t, CompareLetterString);
  }
}
```

(a) Removing Specification (**RS**)

```
bool CompareLetterString(string a, string b){
  var la = a.Where(x => char.IsLetter(x));
  var lb = b.Where(x => char.IsLetter(x));
  return la.SequenceEqual(lb);
}
```

(b) Custom data equality function (**CDE**)

Figure 3.9: Custom data equality example.

Definition 3.23 (Comparison function)**.** *We extend the* ***observe*** *and* ***observeFun*** *constructs with the additional comparison parameter $\xi \in \mathcal{E}$.*

A function $\delta : Loc \to \mathcal{E}$ is called a comparison function, and defined in the following way:

- *For any $\ell :$ `observe` (v, ξ) or $\ell :$ `observeFun` $(\mathbf{f}(v_1, \ldots, v_n), \xi)$ we have $\delta(\ell) = \xi$;*
- *For the observe statements where ξ is left out (not specified), we have $\delta(\ell) = \xi_{def}$.*

We remind the reader that we defined $\mathcal{E}$ and ξ_{def} in Definition 3.6.

In practice the teacher specifies ξ by providing a function $(Val \times Val) \to \{\texttt{true}, \texttt{false}\}$. The teacher can then use ξ to define the equality of *similar computation values*.

The comparison function defined in a specification is then used in matching and trace embedding as defined in Definition 2.13 and discussed in Section 2.3.

We illustrate the usage of a comparison function specified by a teacher on an example.

Example 3.24. *In the previous example we discussed the implementations* ***R3*** *and* ***R4*** *(for the removing strategy); these implementations differ in the character that they use to represent the removed characters.*

Figure 3.9 show the specification ***RS*** *(for the removing strategy), which uses the equality function `CompareLetterString` (**CDE**, given in the same figure). The equality*

```
 1 bool Puzzle(string s, string t) {
 2   if (s.Length != t.Length)
 3     return false;
 4   string cp = t;
 5   for(int i=0; i<s.Length; i++) {
 6     char k = s[i];
 7     bool found = false;
 8     for(int j=0; j<cp.Length; j++) {
 9       if (cp[j] == k) {
10         if (j == 0) {
11           cp = (Char)0+cp.Substring(1)a;
12         } else if(j == cp.Length - 1) {
13           cp = cp.Substring(0, j)+(Char)0;
14         } else {
15           cp = cp.Substring(0, j) +
16                             (Char) 0 + cp.Substring(j + 1);
17         }
18         found = true;
19         break;
20       }
21     }
22     if (!found)
23       return false;
24   }
25   return true;
26 }
```

Figure 3.10: Removing / Separate Computation (**R5**).

function compares only the letters of two strings, while ignoring any other non-letter characters.

The teacher uses the equality function in the observe statement at line 13. Hence, the teacher defines value representation of the both implementations [2] *as equal.*

3.4.2 One-to-many Mapping

According to definition of Trace Embedding (Definition 2.6), an embedding witness π maps one implementation location to a specification location, i.e., it constructs a *one-to-one mapping*. However, it is possible that a student splits a computation of some value over multiple locations; we discuss this on an example.

Example 3.25. *Figure 3.10 gives another example of a student attempt (**R5**) that implements the removing strategy for the Anagram problem.*

In this implementation the student removes a character from the input string across three different locations (on lines 11, 13, and 15-16), depending on the location of the removed character in the string.

[2] In this case of the implementations **R3** and **R4**, but in general also for any other implementation that uses non-letter characters to mark the removed characters.

This example requires to map a *single location* from the specification to *multiple locations* in the implementation. For this reason, we extend the notion of trace embedding to *one-to-many* mappings:

Definition 3.26 (One-to-many mapping)**.** *We extend mapping* $\pi : Loc_1 \rightarrow Loc_2$*, to* $\pi : Loc_1 \rightarrow 2^{Loc_2}$*, with restriction* $\pi(\ell) \cap \pi(\ell') = \emptyset$*, for all* $\ell, \ell' \in Loc_1$ *such that* $\ell \neq \ell'$*.*

It is easy to extend the algorithm EMBED to this settings: the potential graph G is also helpful to enumerate every possible *one-to-many* mapping. However, it is costly and unnecessary to search for arbitrary one-to-many mappings. We use heuristics to consider only a few one-to-many mappings. For example, one of the heuristics in our implementation checks if the same variable is assigned in different branches of an if-statement (e.g., in the above example **R5**, for all three locations there is an assignment to variable *cp*).

We note that although *many-to-many* mappings may seem more powerful, the teacher can always write a specification that is more succinct than the implementation of the student, i.e., the above described one-to-many mappings provide enough expressivity to the teacher.

3.4.3 Non-determinism in Semantics

```
void Puzzle(string s) {
  // ...
  HashSet<char> set =
    new HashSet<char>();
  foreach (char ch in s.Reversed()) {
    set.Add(ch);
  }
  // ...
  foreach (char ch in set) {
    // ...
  }
  // ...
}
```

(a) An implementation.

```
void Puzzle(string s) {
  HashSet<char> set =
    new HashSet<char>();
  foreach (char ch in s) {
    set.Add(ch);
  }
  foreach (char ch in set) {
    observe (ch);
  }
}
```

(b) A specification.

Figure 3.11: Example of two programs whose behavior depends on the iteration order of sets.

Trace Embedding requires *equal values* in the *same order* in both the specification and implementation traces. However, an implementation can use a library function or a data structure with non-deterministic semantics, e.g., the values returned by a random generator or the iteration order over a set data structure.

For such library functions and data-structures we *eliminate non-determinism* by *fixing one particular behavior*. In this way, any implementation becomes *deterministic*, that is, on every execution (on the same input) it generates *the same values in the same order*, as required by the Trace Embedding notion.

For example, we fix the values returned by a random generator or the iteration order over a set during program instrumentation.

We point out that the fixed, deterministic, behavior does not impact functionally correct programs because they cannot rely on some non-deterministic behavior but allow us to apply our matching techniques.

Example 3.27. *Figure 3.11 shows two programs* [3] *(a specification and an implementation) whose behavior depends on the (non-deterministic) order of iteration of sets.* [4]

In the implementation the student adds all the characters of the input string `s` *to the hash-set* `set` *in the reversed order (at lines 5-7), and then iterates this set (lines 9-11).*

In the specification the teacher also adds all the characters of the input string `s` *to the hash-set* `set` *(although not in the reversed order; lines 4-6), and then iterates this set (lines 7-9) and observes the iteration character (line 8).*

The semantics of iteration over sets is under-specified: the order in which the set elements are iterated can depend on the underlying set implementation, an order in which the elements were inserted, and possible other undocumented factors. Hence, the implementation might or might not match the specification, depending on this non-deterministic behavior.

However, the sets in the implementation and the specification are equal (they contain the characters of the input string), and we would want that the implementation matches the specification. As mentioned above, we achieve this by fixing the order in which the set elements are iterated; in this example we might always iterate the characters in the alphabetic order.

3.5 Usage Methodology

In this section we discuss how we envision the described approach would be used by a teacher in a real programming class.

General setting We imagine two usage scenarios:

- A *classroom* setting, where the teacher wants to classify student attempts in order to grade them, and provide a feedback with the grade. Additionally, the teacher can allow students to submit a new version after the feedback has been given.

[3]We point out that these are artificially made examples to illustrate non-deterministic behavior.
[4]Lines with the comment `//...` in the implementation denote some missing code irrelevant for this example.

- A *MOOC* setting, where students are given automated, interactive, feedback, and they can re-submit until they reach a final version. Of course, the number of re-submissions, and the number of times the feedback can be received, could be limited, or points could be deducted for the provided help.

We point out that in both cases the methodology is identical.

High-level view We remind the reader of the discussion in Section 3.1, and the high-level overview in Figure 3.1: The teacher is maintaining a list of specifications, each for one strategy, and both efficient and inefficient specifications. Each specification has a textual feedback assigned to it. In the case of an inefficient specification it could be an explanation of why the strategy is inefficient and how to make it efficient. In the case of an efficient specification it could either acknowledge that the strategy is efficient, or point to other possible efficient solutions (strategies).

Granularity of feedback The granularity of a feedback depends on the teacher. For example, in a programming problem where sorting the input value is an inefficient strategy, the teacher might not want to distinguish between different sorting algorithms, as they do not require a different feedback. However, in a programming problem where students are asked to implement a sorting algorithm it makes sense to provide different feedback for different sorting algorithms.

Input values Our dynamic analysis approach requires the teacher to associate input values with specifications. These input values should cause the corresponding implementations to exhibit their worst-case behavior; otherwise an inefficient implementation might behave similar to an efficient implementation and for this reason match the specification of the efficient implementation. This implies that trivial inputs should be avoided. We remark that it is easy for a teacher, who understands the various strategies, to provide good input values.

Example 3.28. *Two strings with unequal lengths constitute a trivial input for the counting strategy since each of its three implementations **C1-C3** (Figure 1.1 and Figure 1.4) then exit immediately. Similarly, providing a sorted input for the sorting strategy is meaningless.*

For example, for all specifications in the anagram problem we used the following single input:

```
s = "ddeccdlzzzeefabbcedddeghikeddddddz"
t = "eddccdlzzeghikedddddddzzeefabbceddd"
```

When a student submits an implementation, one of two scenarios can happen:

1. The implementation matches some specification, and the student is presented with the feedback assigned to this specification, either efficient or inefficient. In this

case the student is either done, or changes her implementation according to the provided feedback and resubmits the new implementation (potentially receiving new feedback).

2. The implementation does not match any specification and the teacher is notified that there is an unmatched implementation. In this case the student needs to wait until feedback is available.

Unmatched implementation and Inspection step In the case of an unmatched implementation, the teacher studies the implementation and identifies one of the following reasons why it fails to match some existing specification:

(a) The student has written an implementation of a new algorithmic strategy for which there is no specification yet. In this case the teacher writes a *new specification* for this strategy.

(b) The student has written an implementation of an algorithmic strategy for which there is a specification, but the implementation does not match this specification; that is, the existing specification for the strategy is too specific to capture this implementation. In this case the teacher *refines* the existing specification to also match the new implementation.

This process, that we call an *inspection step*, is repeated for each unmatched implementation. We now describe the *inspection step* in more detail.

(a) New specification The teacher creates a new specification using the following steps:

(a.i) Copy the code of the unmatched specification.

(a.ii) Annotate certain values and function calls with observe statements.

(a.iii) Remove any unnecessary code, that is, not needed by the specification.

(a.iv) Identify input values for the dynamic analysis for matching.

(a.v) Associate a feedback with the specification.

(b) Specification refinement To refine a specification, the teacher first identifies one of the following reasons as to why an implementation fails to match it:

(b.i) The implementation differs in minor details.

(b.ii) The specification observes more values than those that appear in the implementation.

```
bool Puzzle(string s, string t) {
  if (s.Length != t.Length)
    return false;
  foreach (char c in t.ToCharArray()) {
    int index = s.IndexOf(c);
    if (index < 0) return false;
    s = s.Remove(index, 1);
  }
  return true;
}
```

(a) An implementation of a new strategy.

```
void Puzzle(string s, string t) {
  foreach (char c in t) {
    int index = s.IndexOf(c);
    if (index < 0) return;
    s = s.Remove(index, 1)
    observe (s);
  }
}
```

(b) A new specification **RS1**.

```
void Puzzle(string s, string t) {
  for (int i = 0; i < t.Length; ++i) {
    int ni = nd1 ? i : t.Length - i - 1;
    char c = t[ni];
    int index = s.IndexOf(c);
    if (index < 0) return;
    s = s.Remove(index, 1)
    observe (s);
  }
}
```

(c) A refined specification **RS2**.

```
void Puzzle(string s, string t) {
  for (int i = 0; i < t.Length; ++i) {
    if (s == t.Substring(i)) return;
    int ni = nd1 ? i : t.Length - i - 1;
    char c = t[ni];
    int index = s.IndexOf(c);
    if (index < 0) return;
    s = s.Remove(index, 1)
    observe (s);
  }
}
```

(d) A refined specification **RS3**.

Figure 3.12: Examples of unmatched implementations and corresponding new or refined specifications (obtained from the inspection step).

(b.iii) The implementation uses different data representation.

The teacher then performs one of the following steps, respectively:

(b.i) The teacher adds a new non-deterministic choice, and if necessary, observes new values or function calls.

(b.ii) The teacher modifies the specification so that it observes less values (e.g., omits some loop iterations).

(b.iii) The teacher creates or refines a custom data-equality function.

Next we discuss some examples of the inspection step, and specification creation and refinement.

Example 3.29. *We discuss this example on the removing strategy (already discussed in Section 1.4).*

*Assume that there is a new unmatched student implementation, similar to the implementation **R1** (Figure 1.3), and that the teacher has not yet seen an implementation of the removing strategy.*

Hence, the teacher creates a new specification for this strategy. An example of a new specification is ***RS1****, given in Figure 3.12. We comment on the above steps for creating a new specification:*

(a.i) The teacher started from the code of the implementation ***R1****.*

(a.ii) The teacher annotated `s`*, the string from which the characters are removed, with the* **`observe`** *statement on line 6.*

(a.iii) The teacher removed unnecessary code, present in the implementation ***R1****, but not needed in the specification (e.g., the beginning if-then-else statement, call to* `ToCharArray`*).*

(a.iv) The teacher then added some inputs values, or reused the existing ones for the Anagram problem (as discussed above).

(a.v) Finally, the teacher associates feedback with the specification; as mentioned in Section 1.4, this could be: "Use a more efficient data-structure to remove characters"

Next we illustrate specification refinement scenarios.

(b.i) There is a new (unmatched) implementation of the removing strategy where the student iterates the string `t` *from the end, instead of the beginning.*

*The teacher refines the earlier specification (****RS1****) using the non-deterministic Boolean variable. The result of this refinement is the specification* ***RS2*** *(Figure 3.12): the teacher replaces the* **`foreach`** *with* **`for`** *loop (in order to access the index being iterated), and using the non-deterministic variable* $\mathtt{nd}_1$ *(on line 3) decides whether to search for the character to remove from the beginning or the end of the string* `t`*.*

(b.ii) There is a new (unmatched) implementation of the removing strategy where the student performs a check in the loop whether the remaining of the original string (`t`*) is equal to the string from which the characters are removed (*`s`*), and returns immediately in this case.*

*The teacher refines the earlier specification (****RS2****) by also returning (exiting) earlier, i.e., by observing less values. The result of this refinement is the specification* ***RS3*** *(Figure 3.12): the additional equality check and early return are added at line 3.*

(b.iii) There is a new (unmatched) implementation of the removing strategy where the student replaces removed characters with some special characters. This is the situation that we have already discussed in Section 3.4.1; we have also discussed the specification ***RS*** *that uses a custom data-equality function for this purpose (Figure 3.9).*

```
$ ./observer.exe anagram.json CS.cs C3.cs -v

DEBUG: Analyzing C3.cs
DEBUG: NDs: {True}
Loops:
  5:    foreach (var item in s)  (line 4)
Expressions:
  12:   item  (line 4)
Potential:
  1: {{12}}
  2: {}
  4: {}
  5: {}
  7: {}
DEBUG: NDs: {False}
Loops:
  5:    foreach (var item in s)  (line 4)
Expressions:
  6:    s.Split(item)  (line 5)
  7:    s.Split(item).Length  (line 5)
  8:    t.Split(item)  (line 6)
  9:    t.Split(item).Length  (line 6)
  12:   item  (line 4)
Potential:
  1: {{12}}
  3: {{6}, {8}}
  4: {{7}, {9}}
  6: {{6}, {8}}
  7: {{7}, {9}}
  MAP:
    1 -> {12}
    3 -> {6}
    4 -> {7}
    6 -> {8}
    7 -> {9}
    OK!
*MATCH*
```

Figure 3.13: A sample output of the OBSERVER tool.

3.6 Implementation

We have implemented the presented framework, namely the algorithms EMBED and MATCHES, in a tool called OBSERVER. The implementation is in C#, and analyzes C# programs; that is, implementations and specifications are written in the C# language (this is because on PEX4FUN students solve exercises in C#).

Trace Construction We have implemented the trace construction (see Section 3.2.2 and Section 3.2.3) by *instrumenting* (discussed below) the student's implementation, and providing **`observe`** and **`cover`** functions, as well as providing the special non-deterministic Boolean variables (distinguished by the prefix `nd`) to the teacher.

Further, we fixed the iteration order on data-structures where order is non-deterministic (as discussed in Section 3.4.3). This was done by instrumenting the iteration and conversion of unordered data-structures to ordered ones.

We want to point out that the framework is language-independent; that is, it would be quite easy to implement it in any (imperative) language that can be suitably, as discussed above, instrumented to generate a computation trace.

Instrumentation The instrumentation was implemented using Microsoft's *Roslyn compiler framework* [5]. The implementation replaces each sub-expression with code that assigns an unique ID (program location), and remembers its value during an execution; and thus creating a computation trace. We point out that assigning a unique ID to each subexpression in the implementation achieves the same purpose as the three-address code in our formal model (each subexpression has a unique program location and it is recorded in the trace).

Next, we discuss an example when the tool is run on some specification-implementation pair. Additional discussion, further examples, and the tool itself are available on the original publication's experimental evaluation web site [6].

The tool is run on a set of inputs (provided as a *JSON* file), a single teacher specification and on one or more student implementations; additionally the teacher has to specify whether to use partial or full matching. The output is Match or No Match, and some additional (optional) information on why matching succeeded or failed.

Example 3.30. *We discuss the invocation of the tool on the counting specification* ***CS'*** *(Figure 3.5) and the implementation* ***C3*** *(Figure 1.4). This sample invocation is shown in Figure 3.13; we discuss some interesting details.*

At line 1, the tool `observer.exe` *is invoked with the inputs* `anagram.json` *(discussed in Example 3.28), on the specification* `CS.cs` *(**CS'**) and the implementation* `C3.cs` *(**C3**). Additionally, the tool is given the* `-v` *flag, which results in printing some additional (verbose) debug information.*

The tool first explores the case when the non-deterministic variable $\mathtt{nd}_1$ *is* `True` *(lines 4-14 of the output). In this case the tool does not find an embedding, and hence the specification and the implementation do not match; we remind the reader that we have discussed this in more detail in Example 3.16.*

The tool next explores (as discussed in Example 3.16) the case when the non-deterministic variable $\mathtt{nd}_1$ *is* `False` *(lines 15-37). In this case the tool finds an embedding and the specification and the implementation match. The actual embedding is given at lines 30-35; we comment on how to interpret the output next.*

Lines 18-23 list the matched expressions of ***C3*** *(we point out that these are the underlined expression in Figure 1.4), together with their unique IDs. Hence, the output at lines 30-35 encodes the following information:*

[5] `http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx`
[6] `http://forsyte.at/static/people/radicek/fse14/`

- *The first **`observe`** statement in **CS'** (at line 4) matches the expression with ID 12 (at line 4).*
- *The third **`observe`** statement in **CS'** (at line 11) matches the expression with ID 6 (at line 5).*
- *And similarly for all other **`observe`** statements and listed expressions.*

The output also includes:

- *The list of loops (lines 16-17) in the implementation. However, this is only relevant for efficient specifications (discussed in Example 3.21).*
- *The constructed potential graph (lines 24-29) during the running of the* EMBED *algorithm (discussed in Section 2.2).*
- *Additional information (not shown here) about the running of the algorithm. We point the interested reader to the evaluation web site mentioned above for more information.*

3.7 Experimental Evaluation

In this section we describe an experimental evaluation of the implementation of our framework.

3.7.1 Data

The evaluation data, as already discussed Section 1.4, consisted of 3 programming problems that already existed on PEX4FUN, and 21 additional programming problems that we created.

Existing Programming Problems The descriptions of the 3 preexisting problems from PEX4FUN are given in Appendix A.1.1. We have chosen these 3 specific problems because they had a high number of student attempts, diversity in algorithmic strategies and a problem was explicitly stated (for many problems on the PEX4FUN platform students have to guess the problem from failing input-output examples).

Created Programming Problems We have created a new course [7] on PEX4FUN. These problems were assigned as a homework to students in a second year undergraduate course. We created this course to understand performance related problems that *computer science students* make, as opposed to regular PEX4FUN users who might not have previous programming experience. We encouraged our students to write efficient implementations by giving more points for performance efficiency than for mere functional correctness.

[7] `http://pexforfun.com/makingprogramsefficient`

The course consists of the 21 programming problems; the descriptions of these problems are given in Appendix A.1.2.

3.7.2 Evaluation

The aggregated evaluation results are in Table 3.1. Below we discuss the results in detail.

Results from the manual code study We first observe that a large number of students managed to write a functionally correct implementation on most of the problems (column *# correct implementations*). This shows that Pex4Fun succeeded in guiding students towards a correct solution.

Our second observation is that for most problems a large fraction of *correct* implementations is inefficient (column *# inefficient implementations*), especially for the *Anagram* problem: 90%. This shows that although students manage to achieve functional correctness, efficiency is still an issue (recall that in our homework the students were explicitly asked and given extra points for efficiency).

We also observe that for all, except two, problems there is at least one inefficient algorithmic strategy, and for most problems (62.5%) there are several inefficient algorithmic strategies (reported in the column N). *These results highly motivate the need for a tool that can find inefficient implementations and also provide a meaningful feedback on how to fix the problem.*

The questions we were interested in the evaluation are the following:

(1) *Are the proposed specification language and framework precise and expressive enough for providing strategy-based feedback?*

(2) *What is the teacher effort required to provide feedback using the proposed method?*

(3) *What is the performance of the framework?*

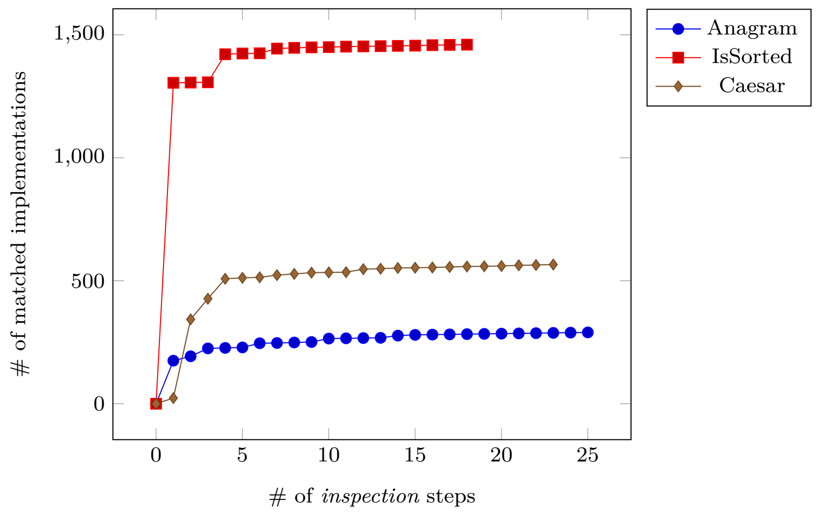
Precision and Expressiveness For each programming problem we used the methodology described in Section 3.5, and wrote a specification for each algorithmic strategy (both efficient and inefficient). We then *manually verified* that each specification matches all implementations of the strategy, hence providing desired feedback for implementations. *This shows that our approach is precise and expressive enough to capture the algorithmic strategy, while ignoring low level implementation details.*

Teacher Effort To provide manual feedback to students the teacher would have to go through every implementation and look at its performance characteristics. In our approach the teacher has to take a look only at a few representative implementations. In column S we report the total number of inspection steps that we required to fully specify one programming problem, i.e., the number of implementations that the teacher would

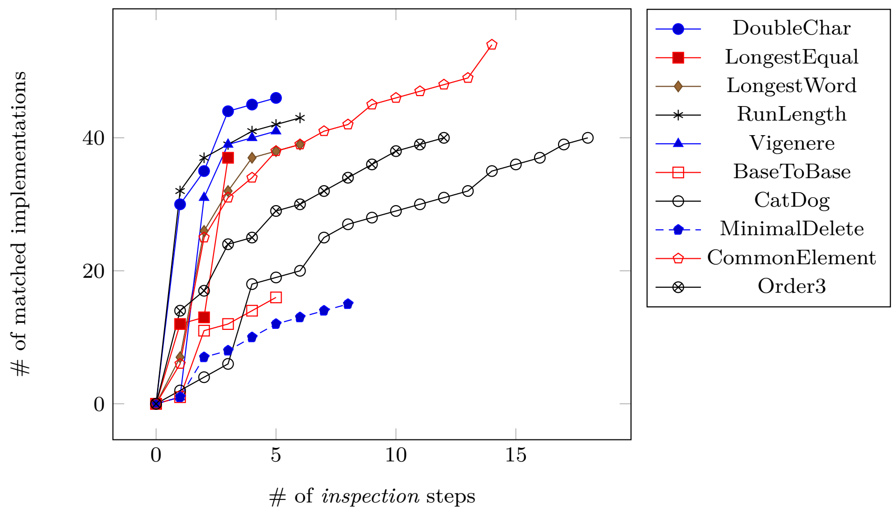
Problem Name	**# correct impls.**	**# inefficient impls.**	N	S	I	ND	$L_S/\overline{L_I}$	O_S	O_I	M	**Performance** (in s) **Avg.**	**Max.**
Anagram	290 (37.9%)	261 (90.0%)	5	25	1	3	1.41	11	89	28357	0.42	7.67
IsSorted	1460 (90.1%)	139 (9.5%)	3	23	2	2	1.45	6	51	13	0.33	1.31
Caesar	566 (81.2%)	343 (60.6%)	5	18	1	1	1.10	7	39	172	0.37	0.83
DoubleChar	46 (97.9%)	31 (67.4%)	1	5	1	0	0.72	3	23	2	0.31	0.42
LongestEqual	37 (78.7%)	1 (2.7%)	1	3	1	0	0.57	1	35	2	0.33	0.44
LongestWord	39 (83.0%)	13 (33.3%)	2	6	2	0	1.31	7	46	15	0.35	0.47
RunLength	43 (97.7%)	32 (74.4%)	1	6	1	0	0.90	8	37	54	0.33	0.44
Vigenere	41 (93.2%)	32 (78.0%)	3	5	1	0	0.64	3	84	6	0.34	0.50
BaseToBase	15 (39.5%)	14 (93.3%)	2	5	1	1	0.35	3	64	13	0.36	0.48
CatDog	41 (87.2%)	8 (19.5%)	2	18	1	1	2.02	21	53	1629	0.36	0.58
MinimalDelete	15 (39.5%)	8 (53.3%)	1	8	2	3	2.21	4	75	10	0.86	4.36
CommonElement	43 (95.6%)	32 (74.4%)	4	14	2	1	0.97	6	79	107	0.36	0.53
Order3	40 (87.0%)	30 (75.0%)	6	12	1	2	1.45	6	78	19	0.40	0.59
2DSearch	37 (84.1%)	36 (97.3%)	3	7	1	1	1.09	2	67	1	0.34	0.45
TableAggSum	11 (25.0%)	10 (90.9%)	1	5	1	1	0.80	3	144	1	0.40	0.53
Intersection	14 (31.8%)	12 (85.7%)	3	7	2	1	0.89	4	73	5	0.37	0.56
ReverseList	39 (97.5%)	0 (0.0%)	0	3	1	0	0.35	4	34	1	0.34	0.44
SortingStrings	41 (91.1%)	34 (82.9%)	5	11	1	1	1.48	13	110	866	0.55	14.59
MinutesBetween	45 (100.0%)	0 (0.0%)	0	5	1	0	0.64	8	101	1	0.37	0.48
MaxSum	42 (95.5%)	17 (40.5%)	2	7	1	1	1.14	2	51	3	0.35	0.47
Median	47 (100.0%)	47 (100.0%)	1	1	1	0	0.39	1	100	1	0.34	0.44
DigitPermutation	36 (100.0%)	1 (2.8%)	1	3	1	0	0.26	4	29	4	0.32	0.44
Coins	27 (65.9%)	14 (51.9%)	2	6	1	1	1.65	4	93	175	2.41	15.44
Seq235	33 (89.2%)	30 (90.9%)	4	12	1	2	1.79	3	232	3	0.94	22.08

Table 3.1: List of all programming problems in the evaluation, with experimental results (N is the number of inefficient strategies; S is the number of inspection steps; I is the number of inputs; ND is the maximal number of used non-deterministic variables; $L_S/\overline{L_I}$ is the largest ratio of specification and average matched implementation (in lines of code); O_s is the maximal number of observed variables in a specification; O_I is the maximal number of observed variables in an implementation; M is the maximal number of mapping functions that our tool had to explore).

(a) # of required inspection steps (1/3)



(b) # of required inspection steps (2/3)



(c) # of required inspection steps (3/3)

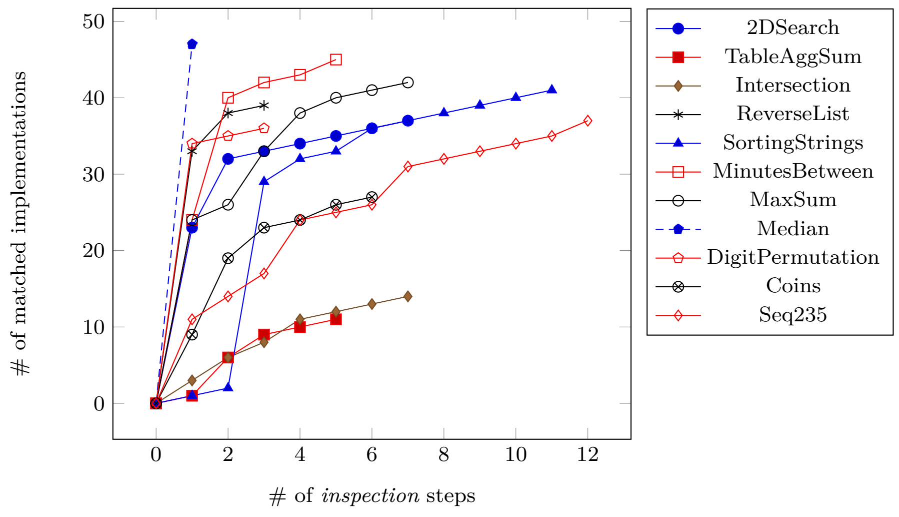
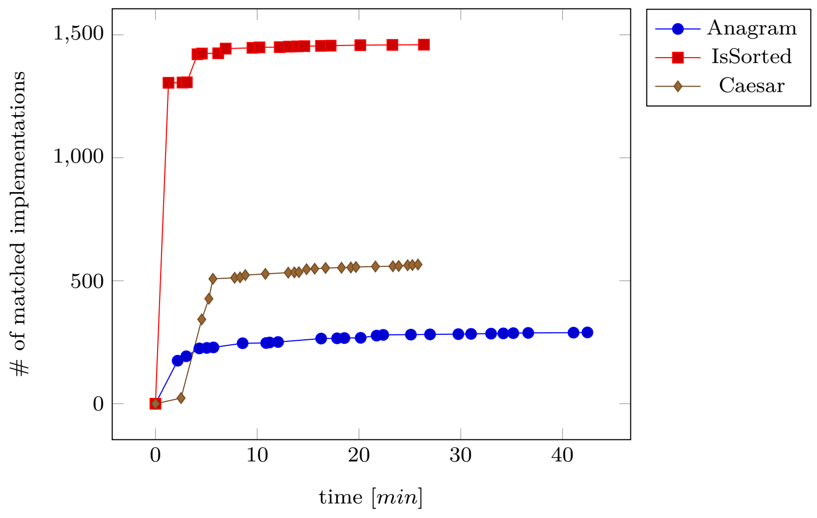
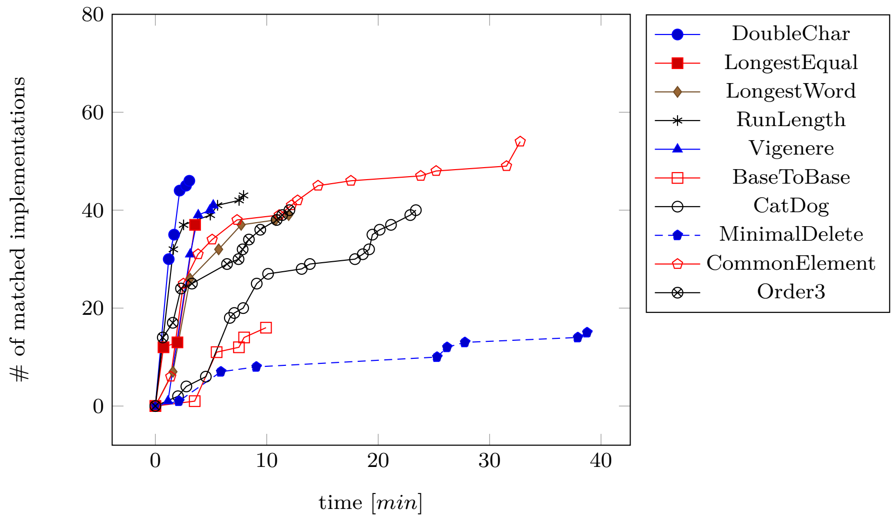


Figure 3.14: The number of inspection steps required to completely specify all the problems.

(a) time required to write/refine specifications (1/3)



(b) time required to write/refine specifications (2/3)



(d) time required to write/refine specifications (3/3)

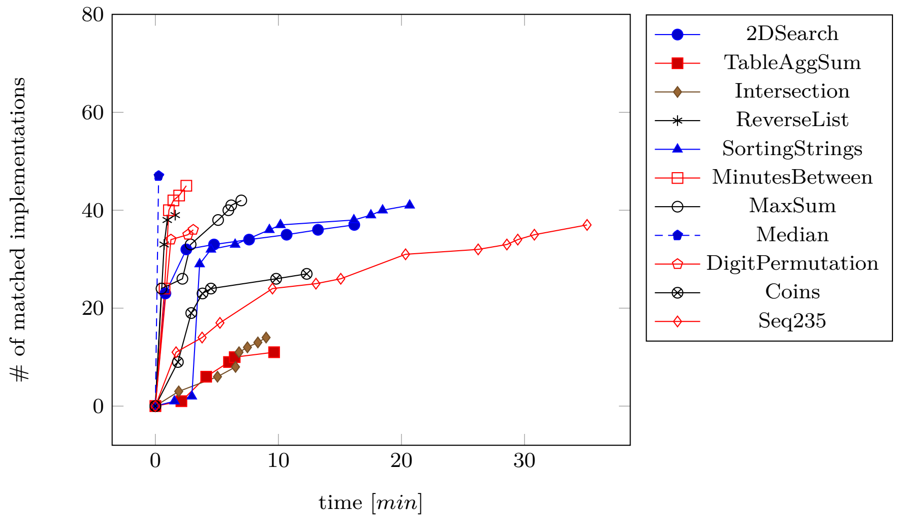


Figure 3.15: Time (in minutes) required to completely specify all the problems.

had to go through to provide feedback on all implementations. For the 3 pre-existing problems *the teacher would only have to go through 66 out of 2316 (or around 3%) implementations to provide full feedback.* Figure 3.14 shows the number of matched implementations with each inspection step, and Figure 3.15 shows the time it took us to create/refine all specifications (we measured the time it takes from seeing an unmatched implementation, until writing/refining a matching specification for it).

In column $^{L_S}/_{L_I}$ we report the largest ratio of specification and average matched implementation in terms of lines of code. We observe that in half of the cases the largest specification is about the same size or smaller than the average matched implementation. Furthermore, the number of the input values that need to be provided by the teacher is 1-2 across all problems (reported in the column I). In all but one problem (*IsSorted*) one set of input values is used for all specifications. Also, in about one third of the specifications there was no need for non-deterministic variables, and the largest number used in one specification is 3 (reported in the column *ND*). *Overall, our semi-automatic approach requires considerably less teacher effort than providing manual feedback.*

The complete log of inspection steps with all intermediate specifications can be downloaded from the original publication's experimental web page [8].

Performance The main purpose of our framework is provide real-time feedback for a large number of students, so performance is critical. Our implementation consists of two parts. The first part is the execution of the implementation and the specification (usually small programs) on relatively small inputs and obtaining execution traces, which is, in most cases, neglectable in terms of performance. The second part is the Embed algorithm. As discussed in Section 2.2 the challenge consists in finding an embedding witness π. With O_S observed variables in the specification and O_I observed variables in the implementation, there are $\frac{O_I!}{(O_I-O_S)!}$ possible injective mapping functions. For example, for the *SortingStrings* problem that gives $\approx 10^{26}$ possible mapping functions ($O_I = 110, O_S = 13$). However, our algorithm reduces this huge search space by constructing a bipartite graph G of potential mappings pairs. In M we report the number of mapping functions that our tool had to explore. For example, for *SortingStrings* only 866 different mapping functions had to be explored. For all values (O_S, O_I and M) we report the maximal number across all specifications.

In the last column we show the *total execution time* required to decide if one implementation matches the specification (average and maximal). Note that this time includes execution of both programs, exploration of all assignments to non-deterministic Boolean variables and finding an embedding witness π. Our tool runs, in most cases, under half a second per implementation. *These results show that our tool is fast enough to be used in an interactive teaching environment.*

[8] `http://forsyte.at/static/people/radicek/fse14/`

3.7.3 Threats to Validity

Unsoundness Our method is unsound in general since it uses a dynamic analysis that explores only a few possible inputs. However, we did not observe any unsoundness in our large scale experiments. If one desires provable soundness, an embedding witness could be used as a guess for a simulation relation [Mil71] that can then be formally verified by other techniques. Otherwise, a student who suspects an incorrect feedback can always bring it to the attention of the teacher.

Program size We evaluated our approach on introductory programming assignments, usually programs with around tens of lines of code. A question might be raised about the applicability of our approach to larger programs (either from advanced education or from real-world software engineering). However, we point out that the goal of the approach was not to analyze arbitrary programs, but rather to develop a framework to help teachers who teach introductory programming with providing performance feedback — this is currently a manual, error-prone and time-consuming task, and hence any advance in this area can bring huge benefits in programming education.

Difficulty of the specification language Although we did not perform a case study with third-party instructors, we report our experience with using the proposed language and the implemented tool; that is, we measure effort taken to provide feedback on problems from a real classroom. We would also like to point out that writing specifications is a one-time investment, which could be performed by an experienced personnel.

3.8 Conclusion

We have presented a novel *semi-automated* approach to performance-related feedback generation in introductory programming. We summarize the key contributions.

Performance feedback generation The key observation is that algorithmic strategies can be identified by observing *key-values* computed during the execution. Following this observation the technical contributions of this chapter are:

- We propose a new language construct, called **`observe`**, that allows a teacher to annotate key-values and thus specify an algorithmic strategy.
- We propose an automated dynamic analysis based approach to test whether a student's implementation matches the teacher's specification.

Implementation and experimental evaluation We describe an implementation of the proposed approach and its experimental evaluation. We show the following (in terms of the criteria set in Section 1.2):

- *Performance:* the feedback is generated under half a second in most cases, hence enabling interactive teaching.
- *Correctness:* we observe no false-positives among generated feedback.
- *Exhaustiveness:* we were able to write specifications for all algorithmic strategies, thus providing feedback on all student attempts.
- *Automation and usefulness:* we show huge savings in teacher effort for providing performance feedback.

CHAPTER 4

Functional Correctness Feedback

In this chapter we describe our approach for providing *repair-based functional* feedback on *incorrect* student attempts.

We start by discussing an overview of the approach on some examples in Section 4.1.

We then continue with the technical development of the approach: in Section 4.2 we describe some preliminaries required to discuss the core algorithms in Section 4.3, and some extensions to it in Section 4.4.

In Section 4.5 we describe the methodology how the approach could be used in practice, we discuss the implementation of the approach in Section 4.6, and in Section 4.7 we describe our experimental evaluation of the implementation and the obtained results.

4.1 Overview of the Approach

In this section we give an overview of our approach for repairing *incorrect student attempts*, in order to provide *functional* feedback.

Our approach to program repair is based on the *matching* relation that we discussed in Chapter 2 (Definition 2.21): two programs match if they have the same memory traces (up to the renaming of the variables by a bijective function).

Given a correct student solution and an incorrect student attempt (also called an *implementation*), the goal is to minimally modify the implementation, such that the correct solution and the modified implementation have the same memory traces (i.e., that they match).

As mentioned earlier, the notion of matching is inspired by the notion of a *simulation relation*, adapted for a dynamic program analysis; hence, we also call this notion *dynamic equivalence* (to stress the usage of dynamic program analysis).

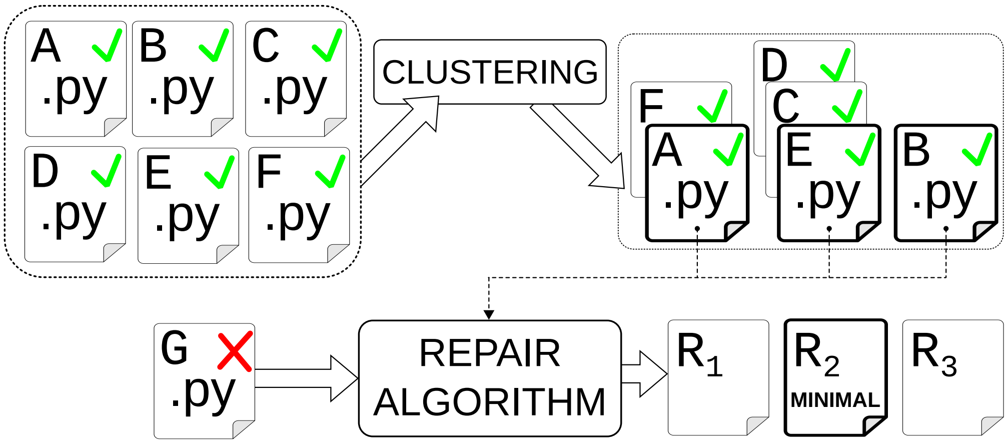


Figure 4.1: High-level overview of the approach.

Figure 4.1 gives a high-level overview of our approach:

(I) For a given programming assignment, we first automatically *cluster the correct student solutions* (**A**-**F** in the figure), based on the notion of *dynamic equivalence* (any two correct solutions in the same cluster match each other). From each cluster (there are three in the figure) we arbitrary pick a *cluster representative* (**A**, **E**, and **B** in the figure).

(II) Given an incorrect *student implementation* (**G** in the figure) we run the *repair algorithm* against each cluster independently; for each cluster the algorithm generates a candidate repair ($\mathbf{R_1}$-$\mathbf{R_3}$ in the figure) such that the repaired implementation and the cluster representative match. Among the candidate repairs, the algorithms selects the *minimal repair* w.r.t. some cost metric [1] ($\mathbf{R_2}$ in the figure). To generate a candidate repair the algorithm combines expressions from *multiple correct solutions* of a cluster.

The overall idea is to use the *wisdom of the crowd*: our approach uses the *already existing correct student solutions* to repair *new incorrect student implementations*. The approach leverages the fact that MOOC courses already have tens of thousands of existing student attempts; this was already noticed by Drummond et al. [Dru+14].

Intuitively, the clustering algorithm groups together similar correct solutions. The repair algorithm can be seen as a generalization of the clustering approach of correct solutions to incorrect attempts. The key motivation behind this approach is as follows: to help the student, with an incorrect implementation, our approach finds the set of most similar correct solutions, written by other students, and generates the smallest modifications that get the student to a correct solution.

[1]In this thesis we use syntactic distance as the cost metric.

We next discuss some key points about the clustering and repair algorithms. We start by discussing the clustering algorithm.

The goal of clustering is twofold:

(1) *Scalability*: elimination of *dynamically equivalent* correct solutions that the repair algorithm would otherwise consider separately.

(2) *Diversity of repairs*: mining of *dynamically equivalent*, but *syntactically different* expressions from the same cluster, which are later used to repair incorrect student attempts.

We next discuss on an example the notion of dynamically equivalent (correct) solutions, and discuss dynamically equivalent expressions later, in context of their usage in the repair algorithm.

Clusters of Dynamically Equivalent Solutions

Two programs, P and Q, belong to the same cluster if they match each other. Hence, clusters are then *equivalence classes* of the matching relation.

In order to show that P and Q match, we need to find a total bijective relation between the variables of P and Q, such that the related variables take the same values, in the same order, during the execution on the same inputs.

We next discuss this notion on an example. All the examples discussed in this section are attempts on the `derivatives` problem: "*Compute and return the derivative of a polynomial function (represented as a list of floating point coefficients). If the derivative is* `0`*, return* `[0.0]`*.*"

Example 4.1. *In Figure 1.7 (Section 1.4) we have given two correct student solutions for the* `derivatives` *problem,* ***D1*** *and* ***D2****.*

Programs ***D1*** *and* ***D2*** *match (are dynamically equivalent) because there is the bijective variable relation:* [2]

$$\tau = \{poly \mapsto poly, deriv \mapsto result, ? \mapsto ?, i \mapsto e, return \mapsto return\}$$

For example, on the input `poly = [6.3, 7.6, 12.14]`*, the variable* `result`*, takes:*

- *the value* `[]` *before the loop,*
- *the sequence of values* `[7.6]`*,* `[7.6,24.28]` *inside the loop, and*

[2]The variables ? and *return* are *special variables* denoting the loop condition and the return value, which we need to make the control-flow and the return values explicit. This is further discussed in the following section (Section 4.2).

- *the value* `[7.6,24.28]` *after the loop.*

Exactly the same values are taken by the variable `deriv`*; and similarly for all the other variables related by* τ*.*

Therefore, ***D1*** *and* ***D2*** *belong to the same cluster, which we denote by* $\mathcal{C}$*. For further discussion, we need to fix one correct solution as a* cluster representative*; we pick* ***D1****, although it is irrelevant which program from the cluster we pick.*

The Repair Algorithm

We have given a high-level overview of the repair algorithm in the beginning of this section. Now we illustrate some details on two examples.

Example 4.2. *In Chapter 1 we discussed the incorrect implementations (on the* `derivatives` *problem)* ***I1*** *(Figure 1.1) and* ***I2*** *(Figure 1.8). We have also given repairs for these two implementations in Figure 1.9.*

These repairs were generated using the cluster $\mathcal{C}$*, with its representative* ***D1****.*

As mentioned above, the top-level procedure also considers other clusters (not discussed here), but found the smallest repair using the cluster $\mathcal{C}$*.*

In the rest of this section we discuss the repair algorithm on a single cluster.

The algorithm generates a repair in two steps:

1. The algorithm generates a set of *local repairs* for each implementation variable (its assigned expression). A local repair either replaces an implementation expression with an expression obtained from the correct cluster, or leaves it unmodified. [3]

2. A (whole-program) repair consist of a combination of local repairs, exactly one for each implementation variable (its assigned expression). There could be numerous such combinations of local repairs; however we are interested in the *correct* repair (such that the repaired implementation and the cluster representative match) with the *smallest cost*. The algorithm finds such a repair by using a constraint-optimization technique.

We are not going to describe these two steps in more detail here (see Section 4.3.2 for a formal definition and a detailed discussion). However, we are going to discuss, on an example, how are expressions obtained from the correct solutions of a cluster, and how are those expressions used in the repair algorithm.

[3]Actually, the algorithm can also delete or introduce new variables or statements. This is discussed in more detail in Section 4.4.

1. `result += [float(poly[e]*e),]`

2.
   ```
   if(e==0): result.append(0.0)
   else: result.append(float(poly[e]*e))
   ```
3. `result.append(1.0*poly[e]*e)`
4. `result.append(float(e*poly[e]))`
5. `result += [e*poly[e]]`

(a) Expressions for the `result` variable inside the loop.

1.
   ```
   if(len(result)==0): return [0.0]
   else: return result
   ```

2.
   ```
   if(len(result)>0): return result
   else: return [0.0]
   ```
3. `return result or [0.0]`

(b) Expressions for the return statement after the loop.

Figure 4.2: Dynamically equivalent expressions.

Example 4.3. *The following expressions are assigned to the variables result (from **D1**) and deriv (from **D2**) inside the loop body; we state the expressions in terms of the variables of the cluster representative **D1**:*

- *`append(result, float(poly[e]*e))`, and*
- *`result + [float(e)*poly[e]]`, where*

*the second expression has been obtained by replacing the variables from **D2** with the variables from **D1** using the variable relation τ (e.g., we have replaced deriv with result, because $\tau(deriv) = result$).*

We call the two expression dynamically equivalent*, because they produce the same values on the same inputs (values for the variables in the expressions).*

In our benchmark we found 15 syntactically different, but dynamically equivalent, ways to write expressions for the assignment to the result variable inside the loop body, and 6 different ways to write the return expression after the loop (observing only different ASTs, and ignoring formatting differences).

*Some of these examples are given in Figure 4.2 (all the expressions are stated in terms of the variables of the cluster representative **D1**).*

*Finally, we discuss how are these different expressions used in a repair. In Figure 1.9 (b) we have given a repair for **I2**. The cluster representative and the repaired program match, because there is the following variable relation (we show only a part of the total bijective relation, relevant for this example):*

$$\tau_2 \subseteq \{poly \mapsto poly, result \mapsto result, i \mapsto e\}$$

This repair combines the following expressions from the cluster $\mathcal{C}$:

- *The first modification is generated using the expression from the cluster representative* ***D1****, at line 3:* `range(1, len(poly))`*.*
- *The second modification is generated using the expression (4.) from Figure 4.2 (a):* `result.append(float(e*poly[e]))`*.*
- *The third modification is generated using the expression (3.) from Figure 4.2 (b):* `return result or [0.0]`*.*

In these expressions, the algorithm replaces the variables of ***D1*** *with the variables of* ***I2****, using the variable relation τ_2 (same as above).*

We point out that the algorithm also considered other dynamically equivalent expressions, but picked the above ones, because they give a correct repair with the smallest cost. For example, to generate the second modification, the algorithm considered all dynamically equivalent expressions in Figure 4.2 (a).

4.2 Program Model

In this section we define a program model that captures key aspects of imperative languages (e.g., C, Python). This model allows us to formalize our notions of program matching and program repair.

Definition 4.4 (Expressions)**.** *Let Var be a set of variables, K a set of constants, and O a set of operations. The set of expressions E is built from variables, constants and operations in the usual way. We fix a set of* special variables *$Var^\sharp \subseteq Var$. We assume that $Var^\sharp$ includes at least the variable ?, which we will use to model conditions, and the variable return, which we will use to model return values.*

Definition 4.4 can be instantiated by a concrete programming language. For example, for the C language, K can be chosen to be the set of all C constants (e.g., integer, float), and O can be chosen to be the set of unary and binary C operations as well as library functions. The special variables are assumed to not appear in the original program text, and are only used for modelling purposes.

Definition 4.5 (Program)**.** *A* program *$P = (Loc_P, \ell_{init}, Var_P, \mathcal{U}_P, \mathcal{S}_P)$ is a tuple, where Loc_P is a (finite) set of* locations*, $\ell_{init} \in Loc_P$ is the* initial location*, Var_P is a (finite) set of* program variables*, $\mathcal{U}_P : (Loc_P \times Var_P) \to E$ is the* variable update function *that assigns an expression to every location-variable pair, and $\mathcal{S}_P : (Loc_P \times \{\texttt{true}, \texttt{false}\}) \to (Loc_P \cup \{end\})$ is the* successor function*, which either returns a successor location in Loc_P or the special value end (we assume $end \notin Loc_P$). We drop the subscript P when it is clear from the context.*

We point the reader to the discussion around the semantics below for an intuitive explanation of the program model.

Definition 4.6 (Computation domain, Memory). *We assume some (possibly infinite) set D of values, which we call the computation domain, containing at least the following values:*

1. $\texttt{true}$ *(bool true);*
2. $\texttt{false}$ *(bool false); and*
3. $\bot$ *(undefined).*

Definition 4.7 (Memory). *Let Var be a set of variables. A memory $\sigma : (\mathit{Var} \cup \mathit{Var}') \to D$ is a mapping from program variables to values, where the set $\mathit{Var}' = \{v' \mid v \in \mathit{Var}\}$ denotes the primed version of the variables in Var; let Σ_{Var} denote the set of all memories over variables $\mathit{Var} \cup \mathit{Var}'$.*

Intuitively, the primed variables are used to denote the variable values after a statement has been executed (see the discussion around the semantics below).

Definition 4.8 (Evaluation). *A function $[\![\cdot]\!] : E \to \Sigma \to D$ is an expression evaluation function, where $[\![e]\!](\sigma) = d$ denotes that e, on a memory σ, evaluates to a value d.*

The function $[\![\cdot]\!]$ is defined by a concrete programming language. The function returns the undefined value ($\bot$) when an error occurs during the execution of an actual program.

Definition 4.9 (Program Semantics). *Let $P = (\mathit{Loc}_P, \ell_{init}, \mathit{Var}_P, \mathcal{U}_P, \mathcal{S}_P)$ be a program. A sequence of location-memory pairs $\gamma \in (\mathit{Loc}_P \times \Sigma_{\mathit{Var}_P})^*$ is called a trace. Given some (input) memory ρ, we write $[\![P]\!](\rho) = (\ell_1, \sigma_1) \cdots (\ell_n, \sigma_n)$ if:*

(1) $\ell_1 = \ell_{init}$;

(2) $\sigma_1(v) = \rho(v)$ *for all* $v \in \mathit{Var}_P$;

(3a) $\sigma_j(v') = [\![\mathcal{U}_P(\ell_j, v)]\!](\sigma_j)$, *and*

(3b) $\sigma_{j+1}(v) = \sigma_j(v')$, *and*

(3c) $\ell_{j+1} = \mathcal{S}_P(\ell_j, \sigma_j(?'))$, *for all* $v \in \mathit{Var}_P$ *and* $0 \leq j < n$; *and*

(4) $\mathcal{S}_P(\ell_n, b) = \mathit{end}$, *for any* $b \in \{\texttt{true}, \texttt{false}\}$.

For some trace element $(\ell, \sigma) \in \gamma$ and variable v, $\sigma(v)$ denotes the value of v before the location ℓ is evaluated (the old value of v at ℓ), and $\sigma(v')$ denotes the value of v after the location ℓ is evaluated (the new value of v at ℓ). The definition of $[\![P]\!](\rho)$ then says:

(1) The first location of the trace is the initial location ℓ_{init}.

(2) The old values of the variables at the initial location ℓ_{init} are defined by the input memory ρ.

(3a) The new value of variable v at location ℓ_j is determined by the semantic function $[\![\cdot]\!]$ evaluated on the expression $\mathcal{U}_P(\ell_j, v)$.

(3b) The old value of variable v at location ℓ_{j+1} is equal to the new value at location ℓ_j.

(3c) The next location ℓ_{j+1} in a trace is determined by the successor function $\mathcal{S}_P$ for the current location ℓ_j and the new value of ? at ℓ_j (i.e., $\sigma_j(?')$).

(4) The successor of the last location, ℓ_n, for any Boolean $b \in \{\mathtt{true}, \mathtt{false}\}$, is the end location *end*.

We next discuss some peculiar aspects of our model.

Control-flow modelling We model if-then-else statements differently, depending on whether they contain loops or they are loop-free. In the former case, the branching is modelled, as usual, directly in the control-flow. In the latter case, (loop-free) statements are (recursively) converted to `ite` expressions that behave like a C ternary operator (we discuss this below on an example). Hence, any loop-free sequence of code is treated as a single block, since blocks can include nested if-then-else statements without loops (for example, similar to Beyer et al. [Bey+09]).

We could have picked a different *granularity* of control-flow; e.g., to treat only straight-line code (without branching) as a block. However, this particular granularity enables matching of programs that have different branching-structure, and as a result the repair algorithm is able to generate repairs that involve missing if-then-else statements (see Section 4.3).

Multiple or no assignments According to Definition 4.5, we have *exactly one* assignment (update) for each variable per program location; more precisely, $\mathcal{U}_P$ assigns exactly one expression to any location-variable pair. However, in C or Python programs a variable can be unmodified (not assigned) at some location or assigned multiple times.

The former case can be easily modeled by setting $\mathcal{U}_P(\ell, v) = v$, if the variable v is not assigned at the location ℓ in the program P. The latter case is modeled by rewriting the assignments in the original program such that there is at most one assignment to each variable at any program location. We illustrate this on an example.

Example 4.10. *In the following sequence of assignment the variable* `a` *is assigned twice:* `a = 1; b = a + 1; a = 2;`

This can be rewritten into a semantically equivalent sequence of assignment, where `a` *is assigned only once:* `b = 1 + 1; a = 2;`*.*

Next, we give a complete example of how a concrete program is represented in our model.

Example 4.11. *We show how a concrete program (**D1** from Figure 1.7 in Section 1.1) is represented in our model.*

The set of locations is

$$Loc = \{\ell_{before}, \ell_{cond}, \ell_{loop}, \ell_{after}\}$$

where:

- $\ell_{init} = \ell_{before}$ *is the location before the loop, and the initial location,*
- ℓ_{cond} *is the location of the loop condition,*
- ℓ_{loop} *is the loop body, and*
- ℓ_{after} *is the location after the loop.*

The successor function is given by:

- $\mathcal{S}(\ell_{before}, \texttt{true}) = \mathcal{S}(\ell_{before}, \texttt{false}) = \ell_{cond}$,
- $\mathcal{S}(\ell_{cond}, \texttt{true}) = \ell_{loop}$, $\mathcal{S}(\ell_{cond}, \texttt{false}) = \ell_{after}$,
- $\mathcal{S}(\ell_{loop}, \texttt{true}) = \mathcal{S}(\ell_{loop}, \texttt{false}) = \ell_{cond}$, *and*
- $\mathcal{S}(\ell_{after}, \texttt{true}) = \mathcal{S}(\ell_{after}, \texttt{true}) = end$.

Note that for non-branching locations the successor functions points to the same location for both `true` *and* `false`*.*

The set of variables is

$$Var = \{poly, result, e, return, ?, it\}$$

where it is used to model PYTHON*'s for-loop iterator. An iterator is a sequence whose elements are assigned, one by one, to some variable (e in this example) in each loop iteration. The expression labeling function is given by:*

- $\mathcal{U}(\ell_{before}, result) =$ `[]`,
- $\mathcal{U}(\ell_{before}, it) =$ `range(1,len(poly))`,
- $\mathcal{U}(\ell_{cond}, ?) =$ `len(it)>0`,
- $\mathcal{U}(\ell_{loop}, e) =$ `ListHead(it)`,
- $\mathcal{U}(\ell_{loop}, it) =$ `ListTail(it)`,
- $\mathcal{U}(\ell_{loop}, result) =$ `append(float(poly[e]*e))`,
- $\mathcal{U}(\ell_{after}, return) =$ `ite(result==[], [0.0], result)`.

For any variable v that is unassigned at some location ℓ we set $\mathcal{U}(\ell, v) = v$, i.e., the variable remains unchanged.

*Finally, we state the trace when **C1** is executed on the input $\rho = \{poly \mapsto [6.3, 7.6, 12.14]\}$. We state only defined variables that change from one trace element to the next. Otherwise, we assume the values remain the same or are undefined ($\perp$) (if no previous value existed).*

$$
\begin{aligned}
&[\![\boldsymbol{C1}]\!](\rho) = \\
&\qquad (\ell_{before}, \{poly \mapsto [6.3, 7.6, 12.14], result' = [], i' = 0, it' = [1, 2]\}) \\
&\qquad (\ell_{cond}, \{?' = \texttt{true}\}) \\
&\qquad (\ell_{loop}, \{e' \mapsto 1, it' = [2], i' \mapsto 1, result' \mapsto [7.6]\} \\
&\qquad (\ell_{cond}, \{?' = \texttt{true}\}) \\
&\qquad (\ell_{loop}, \{e' \mapsto 2, it' = [], i \mapsto 3, result' \mapsto [7.6, 24.28]\} \\
&\qquad (\ell_{cond}, \{?' = \texttt{false}\}) \\
&\qquad (\ell_{after}, \{return \mapsto [7.6, 24.28]\})
\end{aligned}
$$

4.3 Algorithms

In this section we formally define the *clustering* and the *repair algorithm.*

4.3.1 Clustering

The clustering is based on the *trace matching* notion (defined in Definition 2.21), with an additional constraint that the programs have the same control-flow.

Hence, informally, two programs match when:

1. The programs have the *same control-flow* (i.e., the same looping structure); and
2. The corresponding variables in the programs take the same values in the same order, or in other words, the programs have the same memory traces (up to the renaming of the variables by a bijective function).

For the following discussion we fix two programs, $P = (Loc_P, \ell_{init_P}, Var_P, \mathcal{U}_P, \mathcal{S}_P)$ and $Q = (Loc_Q, \ell_{init_Q}, Var_Q, \mathcal{U}_Q, \mathcal{S}_Q)$.

Definition 4.12 (Program Structure)**.** *Programs P and Q have the* <u>*same control-flow*</u> *if there exists a bijective function, called* <u>*structural matching*</u>*, $\pi : Loc_Q \to Loc_P$, s.t., for all $\ell \in Loc_Q$ and $b \in \{\texttt{true}, \texttt{false}\}$, $\mathcal{S}_P(\pi(\ell), b) = \pi(\mathcal{S}_Q(\ell, b))$.*

We remind the reader, as discussed in Section 4.2, that we encode any loop-free program part as single control-flow location; as a result we compare only the *looping structure* of the two programs.

We note that both our matching and repair algorithms require the existence of a *structural matching* π between programs. Therefore, in the rest of this chapter we assume that such a π exists between any two programs that we discuss, and assume that $Loc = Loc_P = Loc_Q$ and $\mathcal{S} = \mathcal{S}_P = \mathcal{S}_Q$, since they can be always converted back and forth using π.

Next, we state two definitions that will be useful later on.

Definition 4.13 (Variables of expression)**.** *Let e be some expression, by $\mathcal{V}(e) = \{v \mid v \in e\}$ we denote the set of variables used in the expresison e. We also say that e ranges over $\mathcal{V}(e)$.*

Definition 4.14 (Variable substitution)**.** *Let $\tau : Var_1 \to Var_2$ be some function that maps variables Var_1 to variables Var_2. Given an expression e over variables Var_1, i.e., $\mathcal{V}(e) \subseteq Var_1$, by $\tau(e)$ we denote the expression, which we obtain from e, by substituting v with $\tau(v)$ for all $v \in Var_1$. Note that $\mathcal{V}(\tau(e)) \subseteq Var_2$.*

In the rest of this chapter we call a *bijective function* $\tau : Var_1 \to Var_2$, between two sets of variables Var_1 and Var_2, a **total variable relation** between Var_1 and Var_2.

Next we give a formal definition of matching between two programs. Afterwards, we give a formal definition of matching between two expressions. The definitions involve execution of the programs on a set of inputs.

Definition 4.15 (Program Matching)**.** *Let I be a set of inputs, and let $\gamma_{P,\rho} = [\![P]\!](\rho)$ and $\gamma_{Q,\rho} = [\![Q]\!](\rho)$ be sets of traces obtained by executing P and Q on $\rho \in I$, respectively.*

We say that P and Q match over a set of inputs I, denoted by $P \sim_I Q$, if there exists a total variable relation $\tau : Var_Q \to Var_P$, such that $\gamma_{P,\rho} \sim^{\tau} \gamma_{Q,\rho}$, for all inputs $\rho \in I$, where $\sim^{\tau}$ is defined in Definition 2.21.

We call τ a matching witness.

Intuitively, a matching witness τ defines a way of translating Q to range over variables Var_P, such that P and Q translated with τ produce the same traces.

Given a set of inputs I, $\sim_I$ is an *equivalence relation* on a set of programs $\mathcal{P}$: the *identity* relation on program variables gives a matching witness for *reflexivity*, the *inverse* τ^{-1} of some total variable relation τ gives a matching witness for *symmetry*, and the *composition* $\tau_1 \circ \tau_2$ of some total variables relations τ_1, τ_2 gives a matching witness for *transitivity*.

We remind the reader that the algorithm for finding τ is discussed in Section 2.2 and Section 2.3.

Definition 4.16 (Expression matching)**.** *Let $\Gamma \subseteq (Loc \times \Sigma_{Var_P})^*$ be a set of traces over variables Var_P, and let e_1 and e_2 be two expressions over variables Var_P, at some location $\ell \in Loc$.*

We say that e_1 and e_2 match over a set of traces Γ, denoted $e_1 \simeq_{\Gamma,\ell} e_2$, if $[\![e_1]\!](\sigma) = [\![e_2]\!](\sigma)$, for all $(\sigma, \ell_i) \in \gamma$ where $\ell_i = \ell$, and all $\gamma \in \Gamma$.

Expression matching says that two expressions produce the same values, when considering the memories at location ℓ, in a set of traces Γ. In the following lemma we state that expression matching is equivalent to program matching; this lemma will be useful for our repair algorithm, which we will state in the next section.

Lemma 4.17 (Matching Equivalence)**.** *Let I be a set of inputs, and let $\Gamma_I = \{[\![P]\!](\rho) \mid \rho \in I\}$ be a set of traces obtained by executing P on I. We have the following equivalence: $P \sim_I Q$ witnessed by $\tau : Var_Q \to Var_P$, if and only if, $e_P \simeq_{\Gamma_I,\ell} \tau(e_Q)$, for all $(\ell, v_1) \in Loc \times Var_P$, where $v_1 = \tau(v_2)$, $e_P = \mathcal{U}_P(\ell, v_1)$, and $e_Q = \mathcal{U}_Q(\ell, v_2)$.*

Proof. "$\Rightarrow$": Directly from the definitions. "$\Leftarrow$": By induction on the length of the trace $\gamma = [\![P]\!](\rho)$ for some $\rho \in I$. $\square$

Clustering We define clusters as the equivalence classes of $\sim_I$. For the purpose of matching and repair we pick an arbitrary class representative from the class and collect expressions from all programs in the same cluster:

Definition 4.18 (Cluster)**.** *Let $\mathcal{P}$ be a set of (correct) programs. A* cluster *$\mathcal{C} \subseteq \mathcal{P}$ is an equivalence class of $\sim_I$. Given some cluster $\mathcal{C}$, we fix some arbitrary class representative $P_\mathcal{C} \in \mathcal{C}$.*

We define the set $\mathcal{E}_\mathcal{C}(\ell, v_1)$ of cluster expressions for a pair $(\ell, v_1) \in Loc \times Var_{P_\mathcal{C}}$: $e_1 \in \mathcal{E}_\mathcal{C}(\ell, v_1)$ iff there is some $Q \in \mathcal{C}$ witnessed by $\tau : Var_Q \to Var_{P_\mathcal{C}}$ such that $v_1 = \tau(v_2)$, $e_2 = \mathcal{U}_Q(\ell, v_2)$ and $e_1 = \tau(e_2)$.

Note that it is irrelevant which program from $\mathcal{C}$ is chosen as cluster representative $P_\mathcal{C}$; we just need to fix some program in order to be able to define the expressions $\mathcal{E}_\mathcal{C}$ over a common set of variables $Var_{P_\mathcal{C}}$. We note that by definition the sets of expressions $\mathcal{E}_\mathcal{C}$ have the following property: for all $e_1, e_2 \in \mathcal{E}_\mathcal{C}(\ell, v)$ we have $e_1 \simeq_{\Gamma_I,\ell} e_2$ that is, expressions e_1 and e_2 match.

Example 4.19. *In Example 4.1 we have discussed why the solutions **D1** and **D2** (given in Figure 1.7) match; therefore these two solutions belong to the same cluster, which we denote here by $\mathcal{C}$, and chose **D1** as its representative $P_\mathcal{C}$.*

In Figure 4.2 we also give examples of different dynamically equivalent expressions of assignment to the variable result inside the loop body and the return statement after the loop, respectively. To be more precise, these were examples of sets $\mathcal{E}_\mathcal{C}(\ell_{loop}, result)$ and $\mathcal{E}_\mathcal{C}(\ell_{after}, return)$, respectively.

4.3.2 Repair

As already mentioned in Section 4.1, the high-level idea of our algorithm is to use clusters of correct solutions to repair an incorrect student implementation. This is depicted in Figure 4.3, which shows the top-level repair algorithm, REPAIRTOP. The algorithm takes

```
def RepairTop(Clusters $\mathfrak{C}$, Implementation $P_{impl}$, Inputs $I$):
    $\mathcal{R}$ = map($\lambda\mathcal{C}.$ Repair$(\mathcal{C}, P_{impl}, I)$, $\mathfrak{C}$)
    return $\arg\min_{R\in\mathcal{R}}$ $cost(R)$
```

Figure 4.3: The Top-level Repair Algorithm.

a set of clusters $\mathfrak{C}$, a student implementation P_{impl}, and a set of inputs I, and returns a repair R (we define the repair R precisely below). The algorithm first (at line 2) runs the Repair algorithm on each cluster $\mathcal{C}$ separately [4], which results in a set of repairs $\mathcal{R}$ (each for one cluster $\mathcal{C}$). The algorithm returns (at line 3) a repair with the minimal cost. In the rest of this section we discuss the underlying Repair algorithm that uses a single cluster $\mathcal{C}$ (with its representative $P_\mathcal{C}$) to repair a student implementation P_{impl}, on some set of inputs I. We assume that P_{impl} and $P_\mathcal{C}$ have the same control-flow.

The goal is to *repair* P_{impl}; that is, to modify P_{impl} minimally, w.r.t. some notion of cost, such that the repaired program matches the cluster. More precisely, the repair algorithm searches for a program $P_{repaired}$, such that $P_\mathcal{C} \sim_I P_{repaired}$, and $P_{repaired}$ should be syntactically close to P_{impl}. Therefore, our repair algorithm can be seen as a *generalization of clustering to incorrect programs.*

We first define a version of the repair algorithm that does not change the set of variables, i.e., $Var_{impl} = Var_{repaired}$. In Section 4.4.2 we extend this algorithm to include changes of variables, i.e., we allow $Var_{impl} \neq Var_{repaired}$. In both cases the control-flow of P_{impl} remains the same.

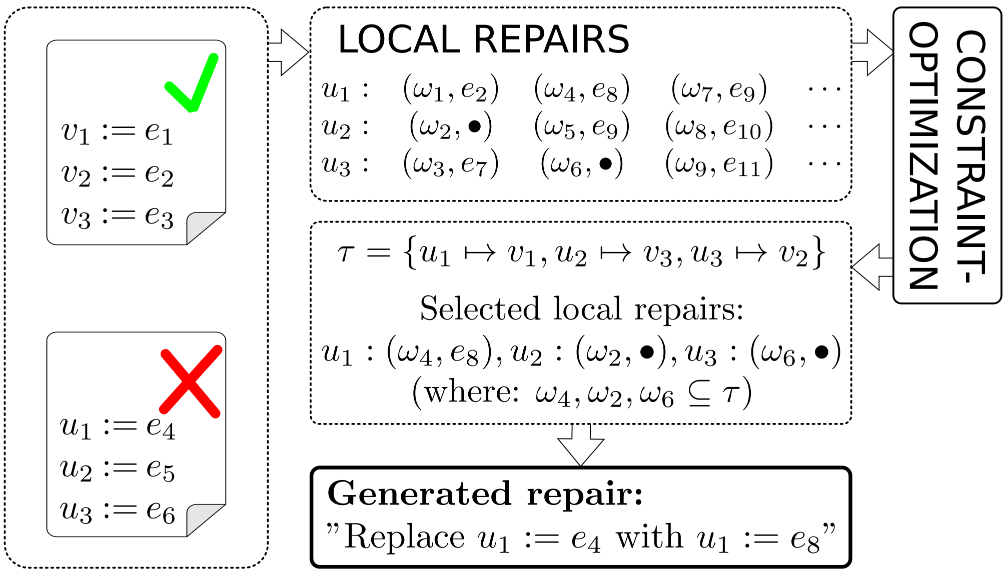
For the following discussion we fix some set of inputs I. Let $\Gamma = \{[\![P_\mathcal{C}]\!](\rho) \mid \rho \in I\}$ be the set of traces of cluster representative $P_\mathcal{C}$ for the inputs I.

High-level idea As we discussed in the previous section, two programs match if all of their corresponding expressions match (see Lemma 4.17). Therefore the high-level idea of our repair algorithm is to consider a set of *local repairs* that modify individual implementation expressions. These local repairs are then combined into a full program repair with the minimal cost, using constraint-optimization techniques. This is depicted in Figure 4.4. We first discuss local repairs; later on we will discuss how to combine local repairs into a full program repair.

Local repairs are defined with regard to *partial variable relations.* It is enough to consider partial variable relations (as opposed to the total variable relations needed for matchings) because these relations only need to be defined for the expressions that need to be repaired.

Definition 4.20 (Local Repair)**.** *Let $(\ell, v_2) \in Loc \times Var_{impl}$ be a location-variable pair from P_{impl}, and let $e_{impl} = \mathcal{U}_{impl}(\ell, v_2)$ be the corresponding expression. Further, let*

[4]This can be done in parallel, as the invocations of the Repair on individual clusters are independent; in fact our implementation (described in Section 4.6) does exactly that.



$v_1 := e_1$
$v_2 := e_2$
$v_3 := e_3$
$u_1 := e_4$
$u_2 := e_5$
$u_3 := e_6$
LOCAL REPAIRS
$u_1 :$ (ω_1, e_2) (ω_4, e_8) (ω_7, e_9) $\cdots$
$u_2 :$ $(\omega_2, \bullet)$ (ω_5, e_9) (ω_8, e_{10}) $\cdots$
$u_3 :$ (ω_3, e_7) $(\omega_6, \bullet)$ (ω_9, e_{11}) $\cdots$
CONSTRAINT-OPTIMIZATION
$\tau = \{u_1 \mapsto v_1, u_2 \mapsto v_3, u_3 \mapsto v_2\}$
Selected local repairs:
$u_1 : (\omega_4, e_8), u_2 : (\omega_2, \bullet), u_3 : (\omega_6, \bullet)$
(where: $\omega_4, \omega_2, \omega_6 \subseteq \tau$)
Generated repair:
"Replace $u_1 := e_4$ with $u_1 := e_8$"

Figure 4.4: High-level overview of the repair algorithm.

$\omega : Var_{impl} \rightharpoonup Var_{P_\mathcal{C}}$ be a partial variable relation such that $v_2 \in dom(\omega)$, let $v_1 = \omega(v_2)$ be the related cluster representative variable, and let $e_\mathcal{C} = \mathcal{U}_{P_\mathcal{C}}(\ell, v_1)$ be the corresponding expression.

A pair $r = (\omega, e_{repaired})$, where $e_{repaired}$ is an expression over implementation variables Var_{impl}, is a local repair for (ℓ, v_2) when $e_\mathcal{C} \simeq_{\Gamma,\ell} \omega(e_{repaired})$ and $\mathcal{V}(e_{repaired}) \subseteq dom(\omega)$. A pair $r = (\omega, \bullet)$ is a local repair for (ℓ, v_2) when $e_\mathcal{C} \simeq_{\Gamma,\ell} \omega(e_{impl})$ and $\mathcal{V}(e_{impl}) \subseteq dom(\omega)$.

We define the cost of a local repair $r = (\omega, e_{repaired})$ as $cost(r) = diff(e_{impl}, e_{repaired})$ and the cost of a local repair $r = (\omega, \bullet)$ as $cost(r) = 0$.

We comment on the definition of a local repair. Let $(\ell, v_2) \in Loc \times Var_{impl}$ be a location-variable pair from P_{impl}, let $e_{impl} = \mathcal{U}_{impl}(\ell, v_2)$ be the corresponding expression, and let r be a local repair for some (ℓ, v_2). In case $r = (\omega, \bullet)$, the expression e_{impl} matches the corresponding expression of the cluster representative under the partial variable mapping ω; this repair has cost zero because the expression e_{impl} is not modified. In case $r = (\omega, e_{repaired})$, the expression $e_{repaired}$ constitutes a modification of e_{impl} that matches the corresponding expression of the cluster representative under the partial variable mapping ω; this repair has some cost $diff(e_{impl}, e_{repaired})$. In our implementation we define $diff(e_{impl}, e_{repaired})$ to be the tree edit distance [Tai79; ZS89] between the abstract syntax trees (ASTs) of the expressions e_{impl} and $e_{repaired}$.

Example 4.21. *We illustrate the notion of local repairs on a few local repairs for* ***I1*** *(Figure 1.1) with regard to the cluster representative* ***D1*** *(Figure 1.7):*

1. *$(\omega_1, \bullet)$ is a local repair for new before the loop (i.e., (ℓ_1, new)), where $\omega_1 = \{new \mapsto result\}$, since the expressions of new and result match (i.e., they take the same values).*

2. *$(\omega_2,$ `if new==[]:  return [0.0] else:  return new`$)$ is a local repair for return after the loop (i.e., $(\ell_4, return)$), where $\omega_2 = \{new \mapsto result, return \mapsto return\}$, since then the return expressions match.*

3. *$(\omega_3,$ `if poly==[]:  return [0.0] else:  return poly`$)$ is a local repair for return after the loop (i.e., $(\ell_1, return)$), where $\omega_3 = \{poly \mapsto result, return \mapsto return\}$, since then the return expressions match.*

Next we discuss how to combine local repairs into a full program repair.

Note. In the definitions below we use notation $r = (\omega, -)$ when $e_{repaired}$ or $\bullet$ is not important in r; similarly we use $r = (-, e_{repaired})$ and $r = (-, \bullet)$ when ω is not important in r.

Definition 4.22 (Repair)**.** *Let R be a function that assigns to each pair $(\ell, v) \in Loc \times Var_{impl}$ a local repair for (ℓ, v). We say that R is consistent, if there exists a total variable relation $\tau : Var_{impl} \to Var_{P_\mathcal{C}}$, such that $\omega \subseteq \tau$, for all $R(\ell, v) = (\omega, -)$. A consistent R is called a repair. We define the cost of R as the sum of the costs of all local repairs: $cost(R) = \sum_{(\ell,v)\in Loc\times Var_{impl}} cost(R(\ell, v))$.*

A repair R defines a repaired implementation $P_{repaired} = (Loc, \ell_{init}, Var_{impl}, \mathcal{U}_{repaired}, \mathcal{S})$, where $\mathcal{U}_{repaired}(\ell, v) = e_{repaired}$ if $M(\ell, v) = (-, e_{repaired})$, and $\mathcal{U}_{repaired}(\ell, v) = \mathcal{U}_{impl}(\ell, v)$ if $M(\ell, v) = (-, \bullet)$, for all $(\ell, v) \in Loc \times Var_{impl}$.

Theorem 4.23 (Soundness of Repairs)**.** *$P_\mathcal{C} \sim_I P_{repaired}$.*

Proof. (Sketch) From the definition of $R(\ell, v_2)$, we have $e_\mathcal{C} \simeq_{\Gamma,\ell} \tau(e_{repaired})$, for all $(\ell, v_2) \in Loc \times Var_{impl}$, where $v_1 = \tau(v_2)$, $e_\mathcal{C} = \mathcal{U}_\mathcal{C}(\ell, v_1)$ and $e_{repaired} = \mathcal{U}_{repaired}(\ell, v_2)$. Then it follows from Lemma 4.17 that $P_\mathcal{C} \sim_I P_{repaired}$. □

Example 4.24. *The repair for example **I1** (Figure 1.9) corresponds to the total variable relation $\tau = \{poly \mapsto poly, new \mapsto result, e \mapsto i, return \mapsto return, ? \mapsto ?\}$. The repair M includes local repairs (1) and (2) from the previous example, where only (2) has cost > 0.*

Next we discuss the algorithm for finding a repair.

The repair algorithm The algorithm is given in Figure 4.5: given a cluster $\mathcal{C}$, an implementation P_{impl}, and a set of inputs I, it returns a repair R. The algorithm has two main parts: First, the algorithm constructs a set of *possible local repairs*; we define and discuss the possible local repairs below. Second, the algorithm searches for a consistent subset of the possible local repairs, which has minimal cost; this search corresponds to solving a constraint-optimization system.

```
1  def Repair(Cluster $\mathcal{C}$, Implementation $P_{impl}$, Inputs $I$):
2      $\pi$ = control-flow matching
3      if $\pi = \emptyset$:
4          return no repair
5      $\Gamma$ = $\{[\![P_\mathcal{C}]\!](\rho) \mid \rho \in I\}$
6      for all $(\ell, v_2) \in Loc \times Var_{impl}$:
7          $LR(\ell, v_2)$ = $\emptyset$
8          $e_{impl}$ = $\mathcal{U}_{impl}(\ell, v_2)$
9          for all $v_1 \in Var_{P_\mathcal{C}}$:
10             $e_\mathcal{C}$ = $\mathcal{U}_{P_\mathcal{C}}(\ell, v_1)$
11             for all $\omega : (\mathcal{V}(e_{impl}) \cup \{v_2\}) \to Var_{P_\mathcal{C}}$ s.t. $\omega(v_2) = v_1$:
12                 if $e_\mathcal{C} \simeq_{\Gamma,\ell} \omega(e_{impl})$:
13                     $LR(\ell, v_2)$ = $LR(\ell, v_2) \cup \{(\omega, \bullet)\}$
14             for all $e \in \mathcal{E}_\mathcal{C}(\ell, v_1)$:
15                 for all $\omega : (\mathcal{V}(e) \cup \{v_1\}) \to Var_{impl}$ s.t. $\omega(v_1) = v_2$:
16                     $LR(\ell, v_2)$ = $LR(\ell, v_2) \cup \{(\omega^{-1}, \omega(e))\}$
17     return FindRepair($LR$)
```

Figure 4.5: The Repair Algorithm.

Definition 4.25 (Set of possible local repairs)**.** *For all $(\ell, v) \in Loc \times Var_{impl}$, we define the set of* <u>*possible local repairs*</u> *$LR(\ell, v)$ as:*

1. *$(\omega, e) \in LR(\ell, v)$, if $\omega(e) \in \mathcal{E}_\mathcal{C}(\ell, \omega(v))$; and*
2. *$(\omega, \bullet) \in LR(\ell, v)$, if $e_\mathcal{C} \simeq_{\Gamma,\ell} \omega(e_{impl})$*

where $e_\mathcal{C} = \mathcal{U}_{P_\mathcal{C}}(\ell, \omega(v))$ and $e_{impl} = \mathcal{U}_{impl}(\ell, v)$.

The set of possible local repairs $LR(\ell, v)$ includes all expressions from the cluster $\mathcal{E}_\mathcal{C}(\ell, \omega(v))$, translated by some partial variable relation ω in order to range over implementation variables. It also includes $(\omega, \bullet)$ if e_{impl} matches $e_\mathcal{C}$ under partial variable mapping ω. Next we describe how the algorithm constructs the set $LR(\ell, v)$ (at lines 6-16).

For the following discussion we fix a pair $(\ell, v_2) \in Loc \times Var_{impl}$ (corresponding to line 6), and some $v_1 \in Var_{P_\mathcal{C}}$ (corresponding to line 9); we set $e_{impl} = \mathcal{U}_{impl}(\ell, v_2)$ and $e_\mathcal{C} = \mathcal{U}_{P_\mathcal{C}}(\ell, v_1)$. Possible local repairs for (ℓ, v_2) are constructed in two steps: In the first step, the algorithm checks if there are partial variable relations $\omega : Var_{impl} \rightharpoonup Var_{P_\mathcal{C}}$ s.t. $e_\mathcal{C} \simeq_{\Gamma,\ell} \omega(e_{impl})$ (at line 11), and in that case adds a pair $(\omega, \bullet)$ to $LR(\ell, v_2)$ (at line 13). In the second step, the algorithm iterates over all cluster expressions $e = \mathcal{E}_{P_\mathcal{C}}(\ell, v_1)$ (at line 14), and all partial variable relations $\omega : Var_{P_\mathcal{C}} \rightharpoonup Var_{impl}$ (at line 15), and then adds a pair $(\omega^{-1}, \omega(e))$ to $LR(\ell, v_2)$ (at line 16). We note that $\omega^{-1}(\omega(e)) = e \in \mathcal{E}_\mathcal{C}(\ell, v_1) = \mathcal{E}_\mathcal{C}(\ell, \omega^{-1}(v_2))$, and thus $(\omega^{-1}, \omega(e))$ is a possible local repair as in Definition 4.25.

We remark that in both steps, the algorithm iterates over all possible variable relations ω. However, since ω relates only the variables of a single expression — usually a small subset of all program variables, this iteration is feasible.

Finding a Repair With the Smallest Cost

Finally, the algorithm uses sub-routine FindRepair (at line 17) that, given a set of possible local repairs LR, finds a repair with smallest cost. FindRepair encodes this problem as a *Zero-One Integer Linear Program (ILP)*, and then hands it to an off-the-shelf ILP solver. Next, we define the ILP problem, describe how we encode the problem of finding a repair as an ILP problem, and how we decode the ILP solution to a repair.

Definition 4.26 ((Zero-One) ILP)**.** *An ILP problem, over variables $\mathcal{I} = \{x_1, \dots, x_n\}$, is defined by an* <u>*objective function*</u> $\mathcal{O} = \boxdot \sum_{1 \leq i \leq n} w_i \cdot x_i$, *and a set of* <u>*linear (in)equalities*</u> $\mathcal{C}$, *of the form* $\sum_{1 \leq i \leq n} a_i \cdot x_i \trianglerighteq b$.[5] *Where* $\boxdot \in \{\min, \max\}$ *and* $\trianglerighteq = \{\geq, =\}$. *A solution to the ILP problem is a variable assignment* $\mathcal{A} : \mathcal{I} \to \{0, 1\}$, *such that all (in)equalities hold, and the value of the objective functions is minimal (resp. maximal) for* $\mathcal{A}$.

We encode the problem of finding a consistent subset of possible local repairs as an ILP problem with variables $\mathcal{I} = \{x_{v_1 v_2} \mid v_1 \in Var_{P_\mathcal{C}} \text{ and } v_2 \in Var_{impl}\} \cup \{x_r \mid r \in LR(\ell, v), (\ell, v) \in Loc \times Var_{impl}\}$; that is, one variable for each pair of variables (v_1, v_2), and one variable for each possible local repair r. The set of constraints $\mathcal{C}$ is defined as follows:

$$\left(\textstyle\sum_{v_2 \in Var_{impl}} x_{v_1 v_2}\right) = 1 \text{ for each } v_1 \in Var_{P_\mathcal{C}} \tag{4.1}$$

$$\left(\textstyle\sum_{v_1 \in Var_{P_\mathcal{C}}} x_{v_1 v_2}\right) = 1 \text{ for each } v_2 \in Var_{impl} \tag{4.2}$$

$$\left(\textstyle\sum_{r \in LR(\ell, v)} x_r\right) = 1 \text{ for each } (\ell, v) \in Loc \times Var_{impl} \tag{4.3}$$

$$-x_r + x_{u_1 u_2} \geq 0 \text{ for each } r = (\omega, -) \in LR \text{ and each } \omega(u_2) = u_1 \tag{4.4}$$

Intuitively, the constraints encode:

(4.1) Each $v_1 \in Var_{P_\mathcal{C}}$ is related to *exactly one* of $v_2 \in Var_{impl}$.

(4.2) Each $v_2 \in Var_{impl}$ is related to *exactly one* of $v_1 \in Var_{P_\mathcal{C}}$. Together (1) and (2) encode that there is a total variable relation $\tau : Var_{impl} \to Var_{P_\mathcal{C}}$.

(4.3) For each $(\ell, v) \in Loc \times Var_{impl}$ *exactly one* local repair is selected.

(4.4) Each selected local repair $r = (\omega, -) \in LR$ is *consistent* with τ, i.e., $\omega \subseteq \tau$.

The objective function $\mathcal{O} = \min\left(\sum_{r \in LR} cost(r) \cdot x_r\right)$ ensures that the sum of the costs of the selected local repairs is minimal.

[5] In general, the coefficients of the objective function and the inequalities range over the real numbers; that is, $w_i, a_i, b \in \mathbb{R}$ (for all $1 \leq i \leq n$). In our encoding, as we discuss below, we have $w_i \in \mathbb{N}$, $a_i \in \{-1, 0, 1\}$, and $b \in \{0, 1\}$.

Decoding an ILP solution to a repair Let $\mathcal{A} : \mathcal{I} \to \{0, 1\}$ be a solution of the ILP problem.

We obtain the following total variable relation from $\mathcal{A}$:

$$\tau(v_2) = v_1 \text{ if and only if } \mathcal{A}(x_{v_1 v_2}) = 1$$

For each $\mathcal{A}(x_r) = 1$, where $LR(\ell, v) = r$, we set $R(\ell, v) = r$.

4.4 Extensions

In this section, we discuss useful extensions to the core material presented so far. These extensions are part of our implementation, but we discuss them separately to make the presentation easier to follow.

4.4.1 Order of Statements

In our model we tacitly assumed that all variables at some location are updated simultaneously. However, in C or Python programs an assignment to some variable can *depend* on the value of some other variable already assigned at the same location (block). Hence, we need to model the order in which variables as assigned in the original program.

Order of statements in the program model To model variable order (or their dependencies) we use primed versions of variables. That is, given some variable v, we denote by its primed version v' the value of v already assigned at that location. More precisely, given some sequence of assignments $v_1 := e_1; \cdots; v_n := e_n$ (in the original program), we replace all occurrences of v_i with v'_i in e_j when $i < j$.

We illustrate this on an example.

Example 4.27. *The following sequence of statements (taken from one of our benchmarks)*

```
f = f + i; i = n1 + 1; n1 = i;
```

is represented in our model (at some location ℓ) as:

- $\mathcal{U}(\ell, f) =$ `f + i`*,*
- $\mathcal{U}(\ell, i) =$ `n1 + 1`*, and*
- $\mathcal{U}(\ell, n1) =$ `i'` *(note that i is primed here).*

Primed and unprimed variables here denote that:

- *f is assigned* before *i,*

- *i is assigned* before *n1 and*
- *n1 is assigned* after *i*.

Note that primed and unprimed variables align with our definition of the semantics (in Section 4.2): (unprimed) v denotes the *old value* (before the location is executed) and (primed) v' denotes the *new value* (after the location is executed).

Order of statements in repair Next we discuss how ordering of statements affects the repair algorithm. We motivate this by considering some repairs for the program discussed in the previous example above.

Example 4.28. *Consider a repair that changes n1 from i' to i, that is to, moves the assignment to n1 before the assignment to i.*

However, the unmodified assignment to i (n1+1) requires that i is assigned before n1. Obviously, both is not possible.

We now discuss how we augment the repair algorithm to consider only the *possible* (valid) order of statements. We first define *conflicting* variable orders.

Definition 4.29 (Variable Order)**.** *Let v be a variable, and e an expression; we define*

$$order(v, e) = \{(v, v_2) \mid v_2 \in e\} \cup \{(v_2, v) \mid v_2' \in e\}$$

*Two orders o_1 and o_2 are **conflicting** if there are two* distinct *variables v_1 and v_2 for which we have both $(v_1, v_2) \in o_1$ and $(v_2, v_1) \in o_2$.*

Let r_1 and r_2 be two local repairs for some pairs $(\ell, v_1) \in Loc \times Var_{impl}$ and $(\ell, v_2) \in Loc \times Var_{impl}$. Further, let e_1 and e_2 be expressions corresponding the repairs r_1 and r_2 (by corresponding we mean $e_i = e$ when $r_i = (-, e)$, or $e_i = \mathcal{U}_{impl}(\ell, v_i)$ when $r_i = (-, \bullet)$). Finally, let $o_1 = order(v_1, e_1)$ and $o_2 = order(v_2, e_2)$ be the corresponding orders.

If o_1 and o_2 are *conflicting orders* we disallow that both local repairs r_1 and r_2 appear in a repair generated by the algorithm.

ILP encoding The restriction can be done in our ILP encoding in the following way: for each r_1 and r_2 with conflicting orders o_1 and o_2 (as just discussed above) we add the following constraint to the ILP model:

$$x_{r_1} + x_{r_2} \leq 1$$

This ensures that the final repair cannot involve both the local repairs r_1 and r_2.

However, we noticed that adding all possible order-constraints to an ILP instance considerably increases the solving time, and also that repairs with conflicting orders rarely occur. Therefore, we adopt the following *incremental* approach, instead of adding all the constraints:

1. When a repair is generated we check for conflicting orders;
2. If there are conflicting orders, then we add constraints to an ILP model only for those orders, and solve a new model.

4.4.2 Adding and Deleting Variables

The repair algorithm described so far does not change the set of variables, i.e., $Var_{repaired} = Var_{impl}$. However, since the repair algorithm constructs a bijective variable relation, this only works when the implementation and cluster representative have the same number of variables, i.e., $|Var_{P_{impl}}| = |Var_{P_{\mathcal{C}}}|$. Hence, we extend the algorithm to also allow the addition and deletion of variables.

We extend total variable relations $\tau : Var_{impl} \rightarrow Var_{P_{\mathcal{C}}}$ to relations $\tau \subseteq (Var_{impl} \cup \{\star\}) \times (Var_{P_{\mathcal{C}}} \cup \{-\})$. We relax the condition about τ being total and bijective: $\star$ and $-$ can be related to multiple variables or none.

When some variable $v \in Var_{P_{\mathcal{C}}}$ is related to $\star$, that is $\tau(\star) = v$, it denotes that a *fresh variable is added* to P_{impl}, in order to match v.

Conversely, when some variable $v \in Var_{impl}$ is related to $-$, that is $\tau(v) = -$, variable v is *deleted* from P_{impl}, together with all its assignments.

We show examples of these kinds of repairs in Section 4.7.1.

Completeness of the algorithm We point out that with this extension the repair algorithm is *complete* (assuming P_{impl} and $P_{\mathcal{C}}$ have the same control-flow). This is because the repair algorithm can always generate a *trivial repair*: all variables $v_2 \in Var_{impl}$ are deleted, that is $\tau(v_2) = -$ for all $v_2 \in Var_{impl}$; and a fresh variable is introduced for every variable $v_1 \in Var_{P_{\mathcal{C}}}$, that is $\tau(\star) = v_1$ for all $v_1 \in Var_{P_{\mathcal{C}}}$. Clearly, this trivial repair has high cost, and in practice it is very rarely generated, as witnessed by our experimental evaluation in the next section.

4.5 Usage Methodology

In this section we discuss how we envision the described approach would be used by a teacher in a real programming class.

The approach can be used in two different scenarios, where the methodology is mostly identical:

- a *classroom* setting, and
- a *MOOC* setting.

We have already discussed these two scenarios in Section 3.5.

We point out that the overall methodology for the functional feedback approach is simpler than for our performance feedback approach, since the former is *fully automated*, while the latter is semi-automated and requires more of the teacher attention.

The workflow The teacher maintains a list of the correct solutions, or more precisely a list of correct clusters. When a student submits a new attempt, the following scenarios can happen:

1. The submitted attempt is *correct*. In this case the clustering algorithm is run on this correct solution with respect to the existing clusters. There are two possible outcomes:
   (a) The correct solution *matches one of the existing clusters*; in this case the solution is *added to this cluster*.
   (b) The correct solution *does not match any of the existing clusters*; in this case *a new cluster is added* to the existing clusters. The new cluster at this point contains only the new correct solution.
2. The submitted attempt is *incorrect*. In this case the top-level repair procedure is run on the incorrect attempt and all the existing clusters. Again, there are two possible outcomes:
   (a) The repair algorithm found a repair for the incorrect attempt and feedback is generated for the student from this repair. We discuss this further in the next section.
   (b) The repair algorithm could not find a repair for the incorrect attempt. In this case the student cannot be given feedback, but the system could notify the teacher and the teacher could manually write a correct solution (repair the incorrect attempt) and add it to the system in the same manner as described in (1.b). After the teacher has added this new solution, the algorithm will be able to find a repair and the feedback is generated in the same way as in (2.a).

Except for the scenario described in (2.b), the system is fully automated and does not require any manual effort from the teacher. Further, in our experimental evaluation (see Section 4.7) we show that the scenario where we cannot provide feedback automatically occurs *very rarely*.

Correct student solutions As mentioned above, the teacher is maintaining a list of correct student solutions. These correct solutions come from two sources:

I. Already existing correct solutions from the previous course offerings. As we already mentioned in Section 4.1, MOOC courses already have tens of thousands of existing student attempts.

II. New correct solutions obtained during the course offering, as described in scenarios (1.a) and (1.b).

We believe that our approach is best suited for MOOC-like setting where there already exists a lot of correct student solutions.

4.6 Implementation

```
$ clara match D1.py D2.py --entryfnc computeDeriv --args "[[[4.5]],
    ↪ [[1.0,3.0,5.5]]]" --verbose 1 --ignoreio 1

// Truncated debug output
[debug] Match: {'computeDeriv': {'$cond': '$cond',
                  '$ret': '$ret',
                  'e': 'i',
                  'ind#0': 'ind#0',
                  'iter#0': 'iter#0',
                  'poly': 'poly',
                  'result': 'deriv'}}
Match!

$ clara feedback D1.py I1.py --entryfnc computeDeriv --args "[[[4.5]],
    ↪ [[1.0,3.0,5.5]]]" --verbose 1 --ignoreio 1

// Truncated debug output
[debug] mapping: {'iter#0': 'iter#0', 'ind#0': 'ind#0', 'e': 'i', '$ret': '$ret', '
    ↪ poly': 'poly', 'result': 'new', '$cond': '$cond'}
[debug] repairs: [3-$ret $ret (1.0)]
[debug] total time: 0.033
* Change 'return (0.0 if (new == []) else new)' to 'return ([0.0] if (new == [])
    ↪ else new)' *after* the 'for' loop starting at line 3 (cost=1.0)
```

Figure 4.6: A sample output of the Clara tool.

We implemented the proposed approach in the publicly available tool Clara[6] (for **CL**uster **A**nd **Rep**A**ir**). The tool currently supports programs in the programming languages C and Python, and consists of:

1. Parsers for C and Python that convert programs to our internal program representation;
2. Program and expression evaluation functions for C and Python, used in the matching and repair algorithms;
3. Matching algorithm;
4. Repair algorithm;
5. Simple feedback generation system.

[6] https://github.com/iradicek/clara

We use the *lpsolve* [7] *ILP* solver, and the *zhang-shasha* [8] tree-edit-distance algorithm.

Feedback generation We have implemented a simple feedback generation system that generates the location and a textual description of the required modifications (similar to, for example, AutoGrader). Other types of feedback can be generated from the repair as well, and we briefly discuss it in Section 6.2.

Next, we discuss some example invocations of the tool; a more detailed documentation can be found on the tool web site (mentioned above).

Example 4.30. *Figure 4.6 shows two invocations of the* Clara*:*

1. *To demonstrate the matching algorithm on the correct solutions **D1** and **D2** (Figure 1.7);*

2. *To demonstrate the repair algorithm and feedback generation on the correct solution **D1** and the incorrect attempt **I1** (Figure 1.9).*

At line 1 Clara *is invoked to test for matching between **D1** and **D2**; the additional arguments are:* `--entryfnc` *to specify the entry function,* `--args` *to provide the inputs,* `--verbose` *to output extra debug information, and* `--ignoreio` *to ignore the standard input and output.* [9] *Lines 4-10 show the matching witness found by the algorithm. We point out that the tool uses two variables (*`ind#0` *and* `iter#0`*) to model* Python*'s for-loop iterator (discussed in Example 4.11); the special variables return and ? (discussed in Section 4.2), are denoted in the tool with* `$ret` *and* `$cond`*.*

At line 13 Clara *is invoked to generate feedback for **I1**, using the correct solution **D1*** [10]*; the arguments are the same as for the matching. Line 16 shows the embedding witness for the matching between **D1** and the repaired **I1**, line 17 shows the repaired variable (*`$ret`*) and the cost (*1.0*, since a single modification has been made:* `0.0` *has been modified to* `[0.0]`*), line 18 shows the time required for the feedback generation, and line 19 shows the generated feedback.*

4.7 Experimental Evaluation

In this section we describe an experimental evaluation of the implementation of our framework. We evaluate our approach in two experiments:

[7]`http://sourceforge.net/projects/lpsolve/`

[8]`https://github.com/timtadh/zhang-shasha`

[9]Without this flag, the tool would introduce additional two variables to model standard input and output, `$in` and `$out`, respectively. However, standard input and output are not relevant for this problem.

[10]This corresponds to running the repair algorithm on a single cluster that contains a single correct solution.

- On a MOOC-size dataset (Section 4.7.1); and
- A user study on repair usefulness (Section 4.7.2).

4.7.1 MOOC-size Experiment

Setup In the first experiment we evaluate Clara on data from the MITx introductory programming MOOC [11], which is similar to the data used in evaluation of AutoGrader [SGSL13].

This data is stripped from all information about student identity, i.e., there are not even anonymous identifiers. To avoid the threat that a student's attempt is repaired by her own future correct solution, we split the data into two sets. From the first (chronologically earlier) set we take only the correct solutions: these solutions are then clustered, and the obtained clusters are used during the repair of the incorrect attempts. From the second (chronologically later) set we take only the incorrect attempts: on these attempts we perform repair. We have split the data in 80 : 20 ratio since then we have a large enough number (12973; see the discussion below) of correct solutions that our approach requires, while still having quite a large number (4293) of incorrect attempts for the repair evaluation. We point out that this is precisely the setting that we envision our approach to be used in: a large number of existing correct solutions (e.g., from a previous offering of a course) are used to repair new incorrect student submissions.

Results The evaluation summary is in Table 4.1; the descriptions of the problems are given in Appendix A.2.1. Clara **automatically** generates a repair for *97.44%* of attempts. As expected, Clara can generate repairs in almost all the cases, since there is always the *trivial* repair of completely replacing the student implementation with some correct solution of the same control flow. Hence, it is mandatory to study the quality and size of the generated repairs. We evaluate the following questions in more detail:

(1) *What are the reasons when* Clara *fails?*

(2) *Does* Clara *generate non-trivial repairs?*

(3) *What is the quality and size of the generated repairs?*

We next discuss the results of the evaluation.

(1) Clara fails Clara fails to generate repair in 110 cases: in 69 cases there are unsupported Python features (e.g., nested function definitions), in 35 cases there is no correct attempt with the same control-flow, and in 6 cases a numeric precision error occurs in the ILP solver. The only fundamental problem of our approach is the inability to generate repairs without matching control-flow; however, since this occurs very rarely,

[11] `https://www.edx.org/school/mitx`

Problem name	**LOC median**	**AST size median**	**# correct**	**# clusters** (% of # correct)	**# incorrect**	**# repaired** (% of # incorrect)		**avg. (median) time** in s	
						Clara	AutoGrader	Clara	AutoGrader
`derivatives`	14	33	1472	532 (36.14%)	481	472 (98.13%)	235 (48.86%)	4.9s (4.4s)	6.6s (5.2s)
`oddTuples`	10	25	9001	454 (5.04%)	3584	3514 (98.05%)	576 (16.07%)	3.0s (2.6s)	25.5s (13.3s)
`polynomials`	13	25	2500	234 (9.36%)	228	197 (86.40%)	17 (7.46%)	1.9s (1.6s)	4.3s (4.0s)
Total	11	25	12973	1220 (9.40%)	4293	4183 (97.44%)	828 (19.29%)	3.2s (2.7s)	19.7s (6.3s)

Table 4.1: List of the problems with evaluation results for the MOOC data (with AutoGrader comparison).

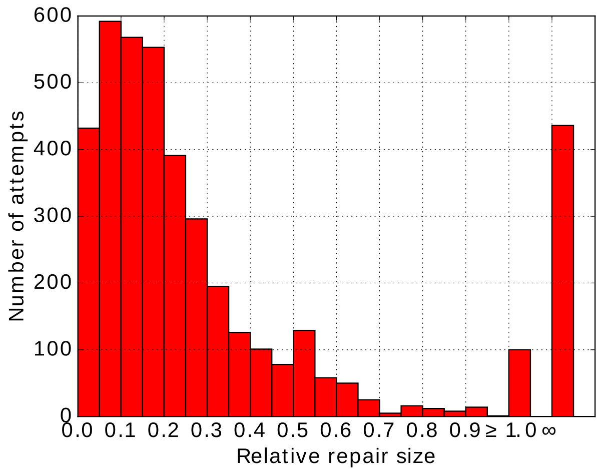


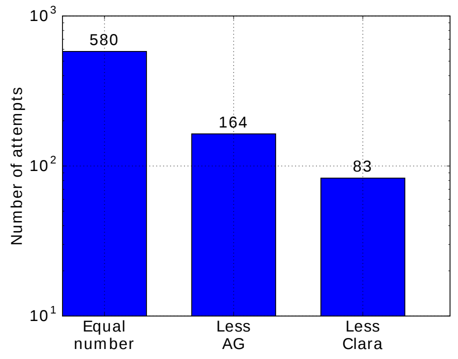
Figure 4.7: Histogram of relative repair sizes.

we leave the extension for future work. *Hence, we conclude that CLARA can repair almost all programs.*

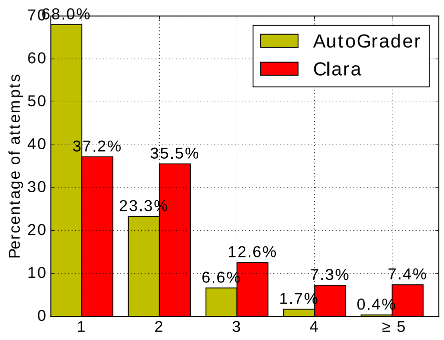
(2) Non-trivial repairs Since a correct repair is also a *trivial* one that *completely replaces* a student's attempt with a correct program, we measure *how much a repair changes the student's program.* To measure this we examine the *relative repair size*: the *tree-edit-distance* of the repair divided by *the size of the AST* of the program. Intuitively, the tree-edit-distance tells us how many changes were made in a program, and normalization with the total number of AST nodes gives us the *ratio of how much of the whole program changed.* Note, however that this ratio can be > 1.0, or even ∞ if the program is empty. Figure 4.7 shows a histogram of relative repair sizes. We note that 68% of all repairs have relative size < 0.3, 53% have < 0.2 and 25% have < 0.1; the last column (∞) is caused by 436 completely empty student attempts. As an example, the two repairs in Figure 1.9 (a) and (b) (discussed in Section 1.1), have relative sizes of 0.03 and 0.24; below, we also state the relative size of some example repairs that we discuss. *We conclude that* Clara *in almost all cases generates a non-trivial repair that is not a replacement of the whole student's program.*

(3) Repair quality and repair size We inspected 100 randomly selected generated repairs, with the goal of evaluating their quality and size. Our approach of judging repair quality and size mirrors a human teacher helping a student: the teacher has to guess the student's idea and use subjective judgment on what feedback to provide. We obtained the following results: (a) In 72 cases Clara generates *the smallest, most natural repair*; (b) In 9 cases the repair is almost the smallest, but involves an additional modification that is not required [12]; (c) In 11 cases we determined the repair, although correct, to be

[12]We discuss two examples of this kind of repairs below (Example 4.32 and Example 4.33).



(a) The number of modified expressions per repair.



(b) Distribution of the number of modified expressions per repair.

Figure 4.8: Comparison of the generated repairs size between AutoGrader and Clara.

different from the student's idea; (d) In 8 cases it is not possible to determine the idea of the student's attempt, although Clara generates some correct repair.

For the cases in (d), we found that program repair is not adequate and further research is needed to determine what kind of feedback is suitable when the student is far from any correct solution. For the cases in (c), the set of correct solutions does not contain any solution which is syntactically close to the student's idea; we conjecture that Clara's results in these cases can partially be improved by considering different cost functions which do not only take syntactic differences into account but also make use of semantic information (see the discussion in Section 4.8). However, in 81 cases (the sum of (a) and (b)), Clara *generates good quality repairs. We conclude that* Clara *mostly produces good quality repairs.*

Summary *Our large-scale experiment on the MOOC data-set shows that* Clara *can fully automatically repair almost all programs and the generated repairs are of high quality.*

Clusters Finally, we briefly discuss the correct solutions, since our approach depends on their existence. The quality of the generated repair should increase with the number of clusters, since then the algorithm can generate more diverse repairs. Thus, it is interesting to note that we experienced *no performance issues with a large number of clusters*; e.g., on `derivatives`, with *532* clusters, a repair is generated on average in *4.9s*. This is because the repair algorithm processes multiple clusters in parallel. Nonetheless, clustering is important for *repair quality*, since it enables repairs that combine expressions taken from different correct solutions from the same cluster, which would be impossible without clustering. We found that 2093 (around 50%) repairs were generated using at least two different correct solutions, and 110 (around 3%) were generated using at least three different correct solutions, in the same cluster.

Comparison with AutoGrader

While the setting of AutoGrader [SGSL13] is different (a teacher has to provide an error model, while our approach is fully automated), the same high-level goal (finding a minimal repair for a student attempt to provide feedback) warrants an experimental comparison between the approaches.

Setup and Data We were not able to obtain the data used in AutoGrader's evaluation, which stems from an internal MIT course, because of privacy concerns regarding student data. Hence, we compare the tools on the same MITx introductory programming MOOC data, which we used in the paper for Clara evaluation. This dataset is similar to the dataset used in AutoGrader's evaluation. AutoGrader's authors provided us with an AutoGrader version that is optimized to scale to a MOOC, that is, it has a weaker error model than in the original AutoGrader's publication [SGSL13]. According to the authors, some error rewrite rules were intentionally omitted, since they are too slow for interactive online feedback generation.

Results The evaluation summary is in Table 4.1. AutoGrader is able to generate a repair for *19.29%* of attempts, using *manually* specified rewrite-rules, compared to *97.44% automatically* generated repairs by Clara. (We note that AutoGrader is able to repair fewer attempts than reported in the original publication [SGSL13] due to the differences discussed in the previous paragraph.) As Clara can generate repairs in almost all the cases, these numbers are not meaningful on their own; the numbers are, however, meaningful in conjunction with our evaluation of the following questions:

(1) *How many repairs can one tool generate that other cannot, and what are the reasons when AutoGrader fails?*

(2) *What are the sizes of repairs?*

(3) *What is the quality of the generated repairs, in case both tools generate a repair?*

We next summarize the results of this evaluation.

(1) Repair numbers In all but one case, when Clara fails to generate a repair, AutoGrader also fails. Further, we manually inspected 100 randomly selected cases where AutoGrader fails, and determined that in 77 cases there is a fundamental problem with AutoGrader's approach [13]: The modifications require fresh variables, new statements or larger modifications, which are beyond AutoGrader's capabilities. *This shows that* Clara *can generate more complicated repairs than AutoGrader.*

In the 100 cases we manually inspected we also determined that *in 74 cases* Clara *generates good quality repairs, when AutoGrader fails.*

[13] We discuss several examples of this kind of repairs below (starting with Example 4.31).

(2) Repair sizes We do not report the relative repair size metric for AutoGrader, because we were not able to extract the repair size from its (textual) output. However, Figure 4.8 (a) compares the relation of the number of modified expressions when both tools generate a repair. We note that the number of modified expressions is a weaker metric than the tree-edit-distance, however, we were only able to extract this metric for the repairs generated by AutoGrader. *We conclude that AutoGrader produces smaller repair in around 10% of the cases.*

Figure 4.8 (b) also compares the overall (not just when both tools generate a repair) distribution of the number of changed expression per repair. *We notice that most of AutoGrader's repairs modify a single expression, and the percentage falls faster than in* CLARA*'s case.*

(3) Repair quality Finally, we manually inspected 100 randomly selected cases where both tools generate a repair. In 61 cases we found both tools to produce the same repair; in 19 cases different, although of the same quality; in 9 cases we consider AutoGrader to be better; in 5 cases we consider CLARA to be better; and in 6 cases we found that AutoGrader generates an incorrect repair. *We conclude that there is no notable difference between the tools when both tools generate a repair.*

Examples of repairs from the manual inspections Finally, we discuss some repair examples that we mentioned while describing the different manual inspections:

- Examples when AutoGrader fails, but CLARA is able to generate a repair; in particular we discuss an example that AutoGrader's authors describe as a big conceptual error.
- Two examples when CLARA generates a repair that is almost the smallest, but involves an additional modification that is not necessary.

Example 4.31. *Figure 4.9 shows an incorrect attempt on the* `oddTuples` *problem, and a repair for it generated by* CLARA.

On the other hand, AutoGrader cannot generate a repair for this attempt for several reasons:

- *It requires addition of a fresh variable, which their error model does not support;*
- *It requires addition of two new statements (assignment to* `new_x` *and a return statement), which their error model also does not support;*
- *It requires changing a whole sub-expression* `i.length()` *with a variable.*

The modification (2.) from the generated repair is not possible in AutoGrader (even if we ignore the freshly added variable) because it requires changing an arbitrary expression

```
1 def oddTuples(aTup):
2     tuple=()
3     for i in aTup:
4         if i.length()%2!=0:
5             tuple=(tuple+(i,))
```

The repair generated by Clara:

1. Add a new variable with assignment `new_x = 1` at the beginning of function `oddTuples`.
2. In codition at line 4 change `i.length() % 2 != 0` to `new_x % 2 != 0`.
3. Add assignment `new_x = new_x + 1` inside the loop starting at line 3.
4. Add return statement `return tuple` after the loop staring at line 3.

The relative cost of this repair is 0.28.

Figure 4.9: A big conceptual error and a repair involving addition of a fresh variable.

```
1 def oddTuples(aTup):
2     if len(aTup)== 1:
3        return aTup
4     elif len(aTup)==0:
5          return aTup
6     else:
7         for n in range(1, len(aTup)):
8             if (n-1)%2==0:
9                 print ((aTup)[n-1])
```

The repair generated by Clara:

1. Add a new variable with assignment `new_ans=()` to the beginning of function `oddTuples`.
2. In condition at line 2 change `len(aTup)==1` to `len(aTup)==0`.
3. In condition at line 4 change `len(aTup)==0` to `len(aTup)==1`.
4. In the iterator at line 7 change `range(1, len(aTup))` to `range(0, len(aTup), 2)`.
5. Add assignment `new_ans = new_ans + aTup[n]` inside the loop starting at line 7.
6. Add return statement `return new_ans` after the loop staring at line 7.

The relative cost of this repair is 0.48.

Figure 4.10: Reverse condition branches in the repair.

with some variable. AutoGrader's authors describe this as a big conceptual error in their paper, and also mention that this is one of the biggest challenges for AutoGrader.

Example 4.32. *Figure 4.10 shows another incorrect attempt on the* `oddTuples` *problem, and a repair for it generated by* Clara.

```
1 def oddTuples(aTup):
2     rTup = ''
3     for index in range(0,len(aTup)):
4         rTup += aTup(index)
5         index += 1
6     return rTup
```

The repair generated by Clara:

1. Change assignment `rTup = ''` to `rTup = ()` at line 2.
2. Add assignment `index = 0` in the beginning of function `oddTuples`.
3. In the iterator at line 3, change `range(0, len(aTup))` to `range(0, len(aTup), 2)`.
4. Change assignment `rTup += aTup(index)` to `rTup += (aTup[index],)` at line 4.

The relative cost of this repair is 0.19.

Figure 4.11: An additional statement.

The generated would also be correct if modifications (2.) and (3.) were omitted, but Clara *generates this repair since the closest correct attempt has the branches reversed, i.e., it first examines the case when* `len(aTup)==0`*, and since the repair algorithm requires the same control-flow, it also suggests these modifications. To eliminate these two modifications, we would have to relax our repair algorithm's requirement on control-flow; however it is not clear at the moment how to do that.*

On the other hand AutoGrader cannot repair this attempt for two reasons:

- *It requires addition of a new variable, which is not expressible in its error model;*
- *It requires adding 3 new statements (two assignments to the fresh variable* `new_ans`*, and a return statement), which is also not expressible in its error model.*

Example 4.33. *Figure 4.11 shows yet another incorrect attempt on the* `oddTuples` *problem, and a repair for it generated by* Clara*.*

This repair would be also correct if the modification (2.) is omitted, but, same as in the previous example, because it is present in the correct solution Clara *generates this modification as well.*

This could be handled by performing an additional analysis that would find this statement redundant.

However, AutoGrader did not manage to generate any repair for this example.

4.7.2 User Study on Usefulness

In the second experiment we performed a user study, evaluating Clara in real time. We were interested in the following questions:

Problem	**LOC median**	**# correct** (exist.+study)	**# clusters** (exist.+study)	**# incorr.**	**# feedback** (% of # incorr.)	**# repair feedback** (% of # feedback)	**time** (in s) **avg.**	**time** (in s) **median**	**# grades 1/2/3/4/5**
`Fibonacci sequence`	12	512+84	70 + 17 (14.60%)	572	539 (94.23%)	440 (81.63%)	10.44	8.51	1 / 7 / 9 / 16 / 13
`Special number`	15	358+59	39 + 3 (10.07%)	121	109 (90.08%)	94 (86.24%)	3.77	2.38	2 / 3 / 8 / 9 / 13
`Reverse Difference`	17	342+46	48 + 8 (14.43%)	103	77 (74.76%)	68 (88.31%)	4.39	3.07	4 / 4 / 5 / 3 / 5
`Factorial interval`	14	391+44	56 + 8 (14.71%)	234	232 (99.15%)	185 (79.74%)	3.33	3.17	2 / 5 / 4 / 5 / 13
`Trapezoid`	14	281+41	36 + 15 (15.84%)	143	129 (90.21%)	121 (93.80%)	7.55	4.82	7 / 5 / 7 / 7 / 5
`Rhombus`	21	264+38	73 + 22 (31.46%)	525	417 (79.43%)	192 (46.04%)	9.16	5.35	6 / 9 / 6 / 5 / 3

Table 4.2: List of the problems with evaluation details for the user study.

(1) *How often and fast is feedback generated (performance)?*

(2) *How useful is the generated repair-based feedback?*

Setup To answer these questions we developed a web interface for Clara and conducted a user study, which we advertised on programming forums, mailing lists, and social networks. Each participant was asked to solve six introductory C programming problems, for which the participants received feedback generated by Clara. There was one additional problem, not discussed here, that was almost solved, and whose purpose was to familiarize the participants with the interface. After solving a problem each participant was presented with the question: *"How useful was the feedback provided on this problem?"*, and could select a grade on the scale from 1 ("*Not useful at all*") to 5 ("*Very useful*"). Additionally, each participant could enter an additional textual comment for each generated feedback individually and at the end of solving a problem.

We also asked the participants to assess their programming experience with the question: *"Your overall programming experience (your own, subjective, assessment)"*, with choices on the scale from 1 ("*Beginner*") to 5 ("*Expert*").

The initial correct attempts were taken from an introductory programming course at *IIT Kanpur, India.* The course is taken by around 400 students of whom several have never written a program before. We selected problems from two weeks where students start solving more complicated problems using loops. Of the 16 problems assigned in these two weeks, we picked those 6 that were sufficiently different.

Results Table 4.2 shows the summary of the results; detailed descriptions of all problems are given in Appendix A.2.2. The columns **# correct** and **# clusters** show the number of correct attempts and clusters obtained from:

(a) the existing ESC 101 data (*exist.* in the table), and

(b) during the case study from participants' correct attempts (*study* in the table).

Note 4.34. *The complete data, with all the attempts, grades and textual comments is publicly available* [14].

Performance of Clara Feedback was generated for 1503 (**88.52%**) of incorrect attempts. In the following we discuss the 3 reasons why feedback could not be generated:

1. In 57 cases there was a bug in Clara, which we have fixed after the experiment finished. Then we confirmed that in all 57 cases the program is correctly repaired and feedback is generated (note that this bug was only present in this real-time experiment, i.e., it did not impact the experiment described in the previous section);

[14] `https://forsyte.at/static/people/radicek/pldi18_survey_data.tar.gz`

2. In 43 cases a timeout occurred (set to 60s);

3. In 95 cases a program contained an unsupported C construct, or there was a syntactic compilation error that was not handled by the web interface (Clara currently provides no feedback on programs that cannot be even parsed).

Further, the average time to generate feedback was **8** seconds. *These results show that* Clara *provides feedback on a large percentage of attempts in real time.*

Feedback usefulness The results are based on *191* grades given by *52* participants. Note that problems have a different number of grades. This is because we asked for a grade only when feedback was successfully generated (as noted above, in 88.52% cases), and because some of the participants did not complete the study. The average grade over all problems is **3.4**. *This shows very promising preliminary results on the usefulness of* Clara. However, we believe that these results can be further improved (see Section 4.8).

The participants declared their experience as follows: 22 as 5, 19 as 4, 9 as 3, 0 as 2, and 2 as 1. While these are useful preliminary results, a study with beginner programmers is an important future work.

Note 4.35. *In the case of a very large repair (cost* > 100 *in our study), we decided to show a generic feedback explaining a general strategy on how to solve the problem. This is because the feedback generated by such a large repair is usually not useful. We generated such a general strategy in 403 cases.*

4.7.3 Threats to Validity

Program size We have evaluated our approach on small to medium size programs typically found in introductory programming problems. The extension of our approach to larger programming problems, as found in more advanced courses, is left for future work. Focusing on small to medium size programs is in line with related work on automated feedback generation for introductory programming (e.g., D'Antoni, Samanta, and Singh [DSS16], Singh, Gulwani, and Solar-Lezama [SGSL13], Head et al. [Hea+17]). We stress that the state-of-the-art in teaching is manual feedback (as well as failing test cases); thus, automation, even for small to medium size programs, promises huge benefits. We also mention that our dataset contains larger and challenging attempts by students which use *multiple functions*, *multiple* and *nested loops*, and our approach is able to handle them.

Unsoundness Our approach guarantees only that repairs are correct over a given set of test cases. This is in accordance with the state-of-the-art in teaching, where testing is routinely used by course instructors to grade programming assignments and provide feedback (e.g., for ESC101 at IIT Kanpur, India [Das+16]). When we manually inspected the repairs for their correctness, we did not find any problems with soundness. We believe that this due to the fact that programming problems are small, human-designed problems that have comprehensive sets of test cases.

In contrast to our dynamic approach, one might think about a sound static approach based on symbolic execution and SMT solving. We decided for a dynamic analysis because symbolic execution can sometimes take a long time or even fail when constructs are not supported by an SMT solver. For example, reasoning about floating points and lists is difficult for SMT solvers. On the other hand, our method only executes given expressions on a set of inputs, so we can handle any expression, and our method is fast. Further, our evaluation showed our dynamic approach to be precise enough for the domain of introductory programming assignments. The investigation of a static verification of the results generated by our repair approach is an interesting direction for future work: one could take the generated repair expressions and verify that they indeed establish a simulation with the cluster against which the program was repaired.

4.8 Conclusion

We have presented a novel *fully-automated* approach for generating functional feedback in introductory programming. We summarize the key contributions.

Functional feedback generation The key idea is to use the already existing correct student solutions to repair the incorrect student solutions:

- We propose a novel clustering algorithm that groups dynamically equivalent correct solutions.
- We propose a novel program repair algorithm that extends our clustering algorithm: the repair algorithm searches for a minimal repair w.r.t. some cluster of correct solutions, such that the repaired program is dynamically equivalent to the programs in that cluster.

Implementation and experimental evaluation We describe an implementation of the proposed approach and its experimental evaluation. We show the following (in terms of the criteria set in Section 1.2):

- *Performance:* the feedback is generated in order of seconds, hence enabling interactive teaching.
- *Correctness:* we observe no incorrect repairs in generated feedback.
- *Exhaustiveness:* the feedback is generated for most of the student attempts (for 97.44% of student attempts in the MOOC experiment, and 88.52% of student attempts in the user study).

- *Usefulness:* in the MOOC experiment we perform a manual inspection of the generated repairs, and conclude that the generated repairs are of good quality; we also perform a user study that shows very promising preliminary results on the usefulness of Clara.

CHAPTER 5

Related Work

In this chapter we give an overview of the related work; some of the related work has already been discussed in Chapter 1, here we give a more systematic and detailed discussion. The discussed related work is far from comprehensive; however, we intend our best to discuss all work that has been encountered while working on this thesis.

We categorize the related work as follows, although some of the mentioned approaches can be put in multiple categories:

- directly related to providing feedback in (mostly programming) education (Section 5.1),
- performance analysis (Section 5.2),
- program repair (Section 5.3),
- relational program analysis (Section 5.4).

5.1 Feedback in (programming) Education

There has been a lot of work in the area of generating automated feedback for programming assignments. We mention several of these approaches, categorized as follows (although, same as above, many approaches can be put in multiple categories):

- *debugging* - approaches that help in debugging errors in student programs (Section 5.1.1),
- *trace-based* - approaches that analyze execution traces (Section 5.1.2),
- *non-functional* - approaches that provide feedback on non-functional properties (Section 5.1.3),

- *classification* - approaches whose goal is to classify student attempts (Section 5.1.4),
- *education platforms* - approaches that develop (mostly web-based) platforms where students can solve programming problems (Section 5.1.5),
- *pedagogy* - approaches whose goal is to study pedagogy of generate feedback (Section 5.1.6),
- *other areas* - approaches that provide feedback on areas outside programming (Section 5.1.7).

5.1.1 Debugging

TALUS [Mur87] matches a student's attempt with a collection of teacher's algorithms. It first tries to recognize the algorithm used and then tentatively replaces the top-level expressions in the student's attempt with the recognized algorithm for generating correction feedback.

LAURA [AL80] heuristically applies program transformations to a student's program and compares it to a reference solution, while reporting mismatches as potential errors (they could also be correct variations).

In contrast, we perform trace comparison (instead of source code comparison), which provides robustness to syntactic variations.

5.1.2 Trace-based Analysis

Striewe and Goedicke [SG11] have proposed localizing bugs by presenting full program traces to the students, but the interpretation of the traces is left to the students. They have also suggested automatically comparing the student's trace to that of a sample solution [SG13], for generating more directed feedback. However, the approach misses a discussion of the situation when the student's code enters an infinite loop, or has an error early in the program that influences the rest of the trace. No implementation was reported.

Apex [Kim+16] is a system that automatically generates error explanations for bugs in programming assignments by comparing their execution traces. In contrast, our goal is to also repair incorrect student attempts.

5.1.3 Non-functional Properties

There has been very little work on generating feedback for non-functional properties.

The *ASSYST* [JU97] system uses a simple form of tracing for counting execution steps to gather performance measurements.

The *Scheme-robo* [SMK01] system counts the number of evaluations done, which can be used for very coarse complexity analysis. The authors conclude that better error messages are the most important area of improvement.

We point out that these approaches cannot point the student to the root cause of the performance issues in her program.

5.1.4 Program Classification

CodeWebs [Ngu+14] classifies different AST sub-trees in equivalence classes based on probabilistic reasoning and program execution on a set of inputs. The classification is used to build a search engine over ASTs to enable the instructor to search for similar attempts, and to provide feedback on some class of ASTs.

OverCode [Gla+14] is a visualization system that uses a lightweight static and dynamic analysis, together with manually provided rewrite rules, to group student attempts.

Drummond et al. [Dru+14] propose a statistical approach to classify interactive programs in two categories (*good* and *bad*), without repairing incorrect programs or generating feedback.

CoderAssist [Kal+16] provides feedback on student implementations of dynamic programming algorithms: the approach first clusters both correct and incorrect programs based on their syntactic features; feedback for incorrect program is generated from a counterexample obtained from an equivalence check (using SMT) against a correct solution in the same cluster.

5.1.5 Education Platforms

Ihantola et al. [Iha+10] present a survey of various platforms developed for automated grading of programming assignments. The majority of these efforts have focussed on checking for functional correctness. This is often done by examining the behavior of a program on a set of test inputs. These test inputs can be manually written or automatically generated [Til+13b].

Pex4Fun [Til+13a], and its successor CodeHunt [Til+14] are browser-based, interactive platforms where students solve programming assignments with hidden specifications, and are presented with a list of automatically generated test cases.

Prutor [Das+16] is a cloud-based web application to conduct introductory programming courses. It provides instant counterexample-based feedback to students while solving programming problems.

5.1.6 Pedagogy

This thesis is focused on technical approach to feedback generation, that is, *how* can we generate feedback for students. An orthogonal research direction is focused on pedagogy of generated feedback, that is, *what* kind of feedback should be given to the students. We believe that the latter is not sufficiently understood and that more research should be done in that direction in the future to fully utilize the power of automated feedback generation.

Head et al. [Hea+17] and Suzuki et al. [Suz+17] discuss how useful feedback can be generated from repairs of student attempts.

Yi et al. [Yi+17] explore different automated program repair (APR) approaches in the context of intelligent tutoring systems. They also conclude that further research is required to understand how to generate the most effective feedback for students from these repairs.

5.1.7 Other Education Areas

As mentioned in Chapter 1, techniques for generating automated feedback have been developed also in other areas, outside of introductory programming (although mostly in STEM area):

- Automata theory: Alur et al. [Alu+13] and D'antoni et al. [D'a+15];
- Arithmetic: Andersen, Gulwani, and Popovic [AGP13] and Andersen, Gulwani, and Popovič [AGP14];
- Algebra and geometry: Singh, Gulwani, and Rajamani [SGR12] and Gulwani, Korthikanti, and Tiwari [GKT11];
- Proofs: Ahmed, Gulwani, and Karkare [AGK13].

5.2 Performance Analysis

The Programming Languages and Software Engineering communities have explored various kinds of techniques to generate various performance related feedback for programs. We discuss some automated static performance analysis and dynamic analysis approaches.

5.2.1 Static Analysis

These techniques aim to automatically and statically (without executing the program) determine upper bounds on program execution time. [1] The techniques can be categorized by the type of programs analyzed (imperative and functional) and the underlying technology.

There are variety of different approaches for performance analysis of imperative programs; for example, based on recurrence equations [Alb+12; DLH90; FMH14], template constraints [CHS15], term-rewriting systems [ADLM15; Bro+16], ranking functions [Ali+10], abstract interpretation [GMC09; GZ10; Her+05], abstract program models [Gup+08; SZV14; SZV17], and interactive verification [MKK17].

[1]In general, these techniques could also analyze other types of resources, e.g., memory or energy consumption.

Since the topic of this thesis are not functional programs, we only briefly mention the RAML project [HAH12]; this is a fully automated technique for inferring polynomial bounds on functional programs, based on amortized analysis and the method of potentials.

These approaches aim for automated performance analysis, while in this thesis we are interested in identifying whether or not there is a performance issue and generating feedback based on its root cause. Further, as already discussed in Section 2.1, automated performance analysis is of little help to *introductory* programming students, who struggle with basic programming concepts and do not yet understand performance implications.

However, we believe that these approaches offer great benefits to experienced software engineers for understanding performance of their code; similarly, we believe that these tools would help senior students who already understand computation complexity and performance issues.

5.2.2 Dynamic Analysis

The dynamic analysis approaches were developed for various performance-related feedback; we mentioned three representative approaches known to us.

Goldsmith, Aiken, and Wilkerson [GAW07] use dynamic analysis techniques for empirical computational complexity.

The Toddler [Nis+13] tool finds a specific pattern in programs: computations with repetitive and similar memory-access patterns.

The Cachetor [NX13] tool finds memoization opportunities by identifying operations that generate identical values.

Same as for the static analysis approaches we believe that these techniques could be very useful to experienced engineers, but not very useful to novice students.

On the other hand, we conjecture that our dynamic relational analysis and the idea of comparing program traces to a specification could also be useful in settings outside of programming education: for empirical computational complexity analysis or finding various patterns in programs (see Section 6.2 for further discussion).

5.3 Program Repair

The research on program repair is extensive, we mention some general-purpose repair approaches as well as some approaches related to programming education.

5.3.1 General Purpose

Gopinath, Malik, and Khurshid [GMK11] propose a SAT-based approach for generating likely bug fixes in programs that manipulate structurally complex data.

Könighofer and Bloem [KB11] propose an approach for automated error localization and correction based on symbolic execution and model-based diagnosis for error localization, and template-based corrections of the RHS expressions.

Jobstmann, Griesmayer, and Bloem [JGB05] and Staber, Jobstmann, and Bloem [SJB05] model the localization and correction problem as a game between the environment that provides different inputs, and the system which provides the repairs.

Prophet [LR16] mines a database of successful patches and uses these patches to repair defects in large, real-world applications. SearchRepair [Ke+15] mines a body of code for short snippets that it uses for repair.

Angelic Debugging [Cha+11] is an approach that identifies *at most one* faulty expression in the program and tries to replace it with a correct value; that is, instead of finding a correct replacement expression, it finds a set of values that the corrected expression should take.

Other approaches are based on program mutation [DW10], or genetic programming [Arc08; For+09], combining mutation with crossing operators and choosing repairs based on a fitness function.

These approaches are not suitable for generating feedback in introductory programming education. For example, Yi et al. [Yi+17] show that using automated program repair approaches out-of-the-box in the setting of programming education seems infeasible due to the low repair rate. Similarly, the general purpose program repair techniques discussed in Goues et al. [Gou+15] on the Intro-Class benchmark, either repair a small number of defects (usually $<50\%$) or take a long time (i.e., over one minute).

In contrast, our program repair approach is designed to work well on programs in introductory programming education, for which it can generate complex and large repairs in couple of seconds.

5.3.2 Programming Education

We mention several approaches to program repair in programming education and comment on their relation to the approach proposed in this thesis.

AutoGrader [SGSL13] is a synthesis- and SAT-based technique for repairing student programs. However, as already discussed in Section 1.1, AutoGrader is limited in several ways.

Refazer [Rol+17] learns programs transformations from example code edits made by the students, and then uses these transformations to repair incorrect student submissions. In comparison to our approach, Refazer does not have a cost model, and hence the generated repair is the first one found (instead of the smallest one).

Rivers and Koedinger [RK17] transform programs to a canonical form using semantic-preserving syntax transformations, and then report syntax difference between an incorrect program and the closest correct solution; the paper reports evaluation on loop-less programs. In contrast, our approach uses *dynamic equivalence*, instead of (canonical) syntax equivalence, for better robustness under syntactic variations of semantically equivalent code.

Qlose [DSS16] automatically repairs programs in education based on different program distances. The idea to consider different semantic distances is very interesting, however the paper reports only a very small initial evaluation (on 11 programs), and Qlose is only able to generate small, template-based repairs. We believe that the idea to consider different distances (cost functions) is orthogonal to our approach and that our approach could be extended with different cost functions (see Section 6.2).

Sarfgen [WSS18] is a follow-up work directly inspired by our approach: it repairs an incorrect student attempt by finding the syntactically closest correct solution and reporting syntactic differences; that is, compared to our dynamic analysis approach, it relies more on syntactic methods. Additionally, Sarfgen has a post-processing step that removes unnecessary repairs (by excluding a subset of repairs and running a repaired program to determine the effect of the exclusion), which could be also easily added to Clara.

5.4 Relational Program Analysis

We mention several approaches to relational program analysis.

Translation Validation [PSS98; Nec00] is an approach for checking equivalence between the original program and the program resulting from compilation or optimization, using a simulation-relation between the variables of the two programs in order to demonstrate equivalence; the simulation relation is derived using heuristics and domain knowledge about the performed transformations.

Lahiri et al. [Lah+15] apply automated differential program verification to show that an approximate program does not diverge significantly from a reference implementation.

Radiček et al. [Rad+17], co-authored by the author of this thesis, is a relational cost analysis based on proof systems combining cost analysis and functional properties.

In contrast to these approaches, our dynamic relational analysis guarantees that results hold only over the provided inputs. We believe that using our analysis as a seed for a formal proof might be an interesting research direction (see Section 6.2 for more details).

CHAPTER 6

Conclusions and Future Work

As defined in the aims of this thesis (Section 1.2) we have:

- Studied a large number of student solutions to better understand performance-related issues in introductory programming.
- Developed novel methods for performance and functional feedback generation in introductory programming. The performance feedback approach is *semi-automated*, while the approach for functional feedback is *fully-automated*. Further, these methods have:
  - a rigorous mathematical foundation, and
  - a practical implementation with experimental evidence showing their speed, correctness, exhaustiveness, and usefulness.

In Section 1.5 we summarized the contributions of this thesis; in this chapter we give a more detailed discussion. After that, we conclude the thesis with some limitations of our approach and directions for future work.

6.1 Contributions

6.1.1 Study of Correct Student Solutions

We study a large number of correct student solutions on 21 different programming problems, with the goal of understanding performance problems faced by students in introductory programming classes.

This study surfaced two things:

1. The notion of *algorithmic strategies*, that is, that for providing performance feedback it is sufficient to understand the student's high-level idea that defines the performance characteristics, while ignoring low-level program details.

2. The idea of *key-values*, that is, that the algorithmic strategy employed by a student solution can be identified by observing the values computed during the execution of the solution.

6.1.2 Dynamic Relational Program Analysis

Based on the observations from the study and the idea of *key-values* we develop *a novel dynamic relational program analysis*. The analysis is inspired by the notion of a *simulation relation* [Mil71] and adapted for a dynamic program analysis.

We give two instantiations of the analysis and use them as a core element in our methods for feedback generation.

Overall, we believe that this thesis shows the effectiveness of using dynamic program analysis in the domain of introductory programming education (as demonstrated by our experimental evaluation; see below):

- Because of their simplicity these kinds of analyses generate the result quite fast.

- Although in general unsound, we show that in this domain it is not difficult to specify all the interesting inputs and hence avoid any unsound results.[1]

6.1.3 Performance Feedback

We propose a novel semi-automated approach for generating performance feedback. As discussed earlier this approach is based on the observations from our code study and the algorithmic strategies notion.

In order to allow the teacher to specify algorithmic strategies we propose a new language construct called **`observe`**. The teacher uses this construct to specify key-values that a strategy computes during the execution, for each strategy that she wishes to provide feedback for. The key strength of this language extensions is that it is lightweight: the teacher has to implement the strategy for which she wants provide feedback and merely annotate the expressions that compute the key values with **`observe`**.

We also propose a novel automated algorithm, based on our dynamic relational program analysis, that decides whether a student's implementation matches (implements) the teachers specification.

[1]This does not hold in general for analysis of arbitrary program.

6.1.4 Functional Feedback

We propose a novel automated approach for generating functional feedback. The key idea behind our approach is to use the existing correct student solutions, which are available in tens of thousands in large MOOC courses, to *repair incorrect student solutions*.

The first step of the approach is to cluster existing correct solutions into equivalence clusters, using our dynamic relational program analysis.

The second step is to use our novel repair algorithm to find a minimal repair with respect to the clusters of correct solutions. The key idea of the repair algorithm is to extend our dynamic relation program analysis to incorrect programs. That is, the algorithm searches for the minimal repair such that the repaired program matches one of the correct clusters.

The key strengths of the proposed algorithm are:

- It can generate various kinds of modifications: changing expressions, swapping the order of statements, adding and delimiting variables and statements.
- It is complete modulo the control-flow, i.e., if a correct solution with the matching control-flow as the incorrect solution exists, the algorithm is able to find a repair.

6.1.5 Implementations and Experimental Evaluation

We have implemented both of the proposed approaches in a practical tool and performed an experimental evaluation on a large number of student attempts. We show the following:

- *Performance:* Both of the approaches generate feedback in order of seconds; this makes them suitable for real-time feedback generation (e.g., in context of MOOC courses).
- *Correctness:* In both of the approaches we manually verified that the generated feedback is sound; this is especially important in context of our dynamic program analysis.
- *Exhaustiveness:* Both of the approaches generate feedback in most cases:
  - In the context of performance feedback we evaluate this by the ability to write specifications for all algorithmic strategies, thus providing feedback on all student attempts.
  - In the context of functional feedback we evaluate this by the percentage of incorrect solutions for which the approach is able to generate repairs; in the MOOC experiment the repair is generated for 97.44% of attempts, while in the user study it is generated for 88.52% of attempts.
- *Automation and usefulness:*

- For the performance feedback generation we show huge savings in teacher effort for providing feedback (e.g., for one of the problems we show that teacher would have to examine only 3% of student attempts to provide feedback on all of them).
- For the functional feedback we study the usefulness of the generated repair. In the MOOC experiment we manually check a number of generated repairs and determine that in 81% of cases Clara generates good quality repairs. In the user study the participants graded the generated feedback with the average grade of 3.4 (on a scale from 1 to 5).

6.2 Future Work

In this section we discuss some ideas for the future work.

6.2.1 Using Dynamic Analysis as a Seed for a Formal Proof

Our dynamic analysis is unsound in the same way as is program testing: the results are guaranteed to hold only over the provided inputs.

However, in our experimental evaluation we did not observe any unsoundness. We believe that this is the case because it is easy to provide quite exhaustive set of inputs for the problems in introductory education. [2]

On the other hand, an interesting research direction might be to investigate whether the results of our analysis can be used as a seed for a formal proof. More precisely, an embedding witness or a total variable relation produced by our methods could by used as a guess for a simulation relation that can then be formally verified by other techniques.

6.2.2 Automated Specification Generation

In our approach to performance feedback generation, the teacher needs to write the specifications manually. Although our experimental evaluation (see Section 3.7) shows that writing specifications requires considerably less teacher effort that manual grading, it would be interesting to investigate if this process could be even more automated.

We sketch a possible direction of automation, based on the insight that our approach does not require specifications directly; they are just a way to specify a set of key-value traces that implementations of the corresponding algorithmic strategy should match. Hence, instead of writing a specification for some strategy, the teacher would manually label several implementations of this strategy, and the system would automatically find a set of traces that all of them match. When there is a new implementation that does not match any of the existing strategies, the teacher labels it accordingly and the system

[2]This is especially the case for the teachers with ample experience in the introductory programming courses.

automatically generalizes the traces of the corresponding strategy. Some challenges of this approach are:

- How to guarantee that the generated traces are not too general (so that they match too many programs) or too specific (so that they match too few programs and need to be generalized too often)?
- How can the teacher interpret the automatically generated traces?

6.2.3 Pattern Finding in General-Purpose Programs

We have evaluated our *Trace Embedding* approach on introductory programming assignments because the aims of this thesis were to develop methods to generate feedback in introductory education. However, it would be interesting to investigate whether the similar methods could be used to analyze larger (general-purpose) programs.

Although it is currently not clear how this approach would apply to larger programs, we imagine the following scenarios.

Similar to the Toddler [Nis+13] and Cachetor [NX13] projects (mentioned in Section 5.2), we imagine our approach being used for detecting different inefficiency patterns in code. While these projects look for a set of specific patterns, we imagine that our approach would allow a domain expert to write specifications (similar to the specifications written by the teacher in our approach) that describe inefficient patterns.

Further, in would be interesting to investigate whether this could be used in a similar way for finding patterns going beyond performance, that is, for functional correctness. For example, to find incorrect usages of an API.

6.2.4 Cost Function in the Repair Algorithm

As discussed in Section 4.3, our repair algorithm uses the tree edit distance [Tai79; ZS89] between the abstract syntax trees (ASTs) as the cost function while finding the repair with the minimal cost.

We believe that the cost function could take into account more information; we mention two possible ideas for extension, based on the existing work.

In QLOSE [DSS16] approach the authors are also considering various notions of *semantic distance between programs*, going beyond pure syntactic differences, but taking into account differences in program behavior.

Demyanova, Veith, and Zuleger [DVZ13] study variable roles (e.g., flag, loop iterator, counter, index) and their use in software analysis. In would be interesting to extend CLARA with knowledge of variable roles and reduce the repair search space such that it only matches variables with the same roles. This would not only have performance benefits (on the repair algorithm), but potentially might better match the student's intention.

6.2.5 Control-flow Matching

Our clustering and repair algorithms (Section 4.3) are restricted to the analysis of programs with the same control-flow. In our experiments (Section 4.7) we observed only 35 cases (out of 4293 attempts in our MOOC experiment) that require analysis of programs with different control-flow. Hence, we did not extend our algorithms to handle programs with different control-flow.

We conjecture that the algorithms could be extended to programs with similar (instead of the same) control-flow.

6.2.6 Extension to Functional Programming Languages

Our relational dynamic analysis and both of our feedback generation techniques (for performance and functional properties) are presented and evaluated in the context of imperative programming languages.

We already argued that our techniques should easily apply to other imperative languages, besides the ones we use in our implementations, since the techniques do not depend on any particular feature of the implementation languages.

It would be interesting to investigate if the same techniques could be applied in the context of functional programming languages as well. We speculate that it would not be difficult to obtain an execution trace of a functional program and use the same algorithms.

6.2.7 Pedagogy of Automated Feedback Generation

While this thesis is mainly focused on the technical problem, an interesting orthogonal direction for future work is to consider pedagogical research questions ([Hea+17; Suz+17]). For example, in the context of functional feedback:

1. How much information should be revealed to the student (the line number, an incorrect expression, the whole repair)?
2. Should the use of automated help be penalized?
3. How much do students learn from the automated help?

One possible direction is to consider different types of feedback that can be generated from a repair of a student's program. The straightforward feedback is to present the student with full or partial set of changes. However, different kinds of feedback are possible; for example: A course instructor could annotate variables in the correct solutions with their descriptions, and when a repair for some variable is required, a matching feedback is shown to a student.

APPENDIX A

Problems List

Here we list the descriptions of all programming problems discussed in the thesis.

A.1 Performance Feedback Evaluation

A.1.1 Existing Problems on Pex4Fun

`Anagram`

Given two strings `s` and `t`, determine whether they are anagrams. Two strings are anagrams when one can be permuted (rearranged) to match the other.

`IsSorted`

Given an integer input array `A`, check if `A` is sorted (i.e., whether `A[i]` $\leq$ `A[j]`, for all $i < j$).

`Caesar`

Apply the Caesar cipher [1] to the input string.

A.1.2 Created Course Problems

`DoubleChar`

Double every character in the input string.

For example: return `"xxyyzz"` for the input string `"xyz"`.

[1] `https://en.wikipedia.org/wiki/Caesar_cipher`

LongestEqual

Given an integer input array, find the length of the longest sequence of equal values.

LongestWord

Find the longest word in the input string (words are contiguous sequences of letters separated by spaces).

RunLength

Run-length [2] encode the input string.

Vigenere

Return the Vigenere cipher [3] of the input string.

BaseToBase

Convert the input string, representing a number in the base b_1, to the base b_2, where $2 \leq b_1, b_2 \leq 36$. Characters 0 - 9 represent values 0 - 9 and characters A - Z represent values 10 - 35.

CatDog

Check if the number of `"cat"` substrings in the input string the same as the number of `"dog"` substrings.

MinimalDelete

Find the minimal number of characters that have to be deleted so that the two input strings become equal.

CommonElement

Find the smallest common element of the three sorted input integer arrays.

Order3

Sort an integer array containing only the elements 1, 2, and 3, by using solely the `swap` function.

Note: `swap(A, i, j)` changes the value of the array element `A[i]` with the value of the array element `A[j]`, and vice versa.

[2] http://en.wikipedia.org/wiki/Run-length_encoding
[3] https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher

`2DSearch`

Find the position of an integer in a 2D ordered integer array.

`TableAggSum`

Return the aggregated sum for IDs (the input is an array of ID-value pairs).

`Intersection`

Compute the intersection of the three input integer arrays.

`ReverseList`

Reverse the pointers in a linked list represented as an integer array.

`SortingStrings`

Sort the input array of strings, where each string is four characters long.

`MinutsBetween`

Calculate the number of minutes between two points of time (represented as strings in the `"HH:MM"` 24-hour format).

`MaxSum`

Given an integer array, find the contiguous subarray whose sum is maximal and return that sum.

`Median`

Compute the median of the input integer array.

`DigitPermutation`

Calculate the next permutation of the input integer array with regard to the lexicographic ordering.

`Coins`

For the input array of integers (representing coin values), and the input value S, find the minimum number of coins the sum of which is S.

`Seq235`

For the input n, find the n^{th} element in the sequence of numbers that are divisible only by 2, 3 and 5 (starting with 1,2,3,4,5,6,8, ...).

A.2 Functional Feedback Evaluation

A.2.1 MOOC experiment

`derivatives`

Compute and return the derivative of a polynomial function represented as a list of floats. If the derivative is 0, return [0.0].

input: list of numbers (length ≥ 0).

return: list of numbers (floats).

`oddTuples`

input: a tuple `aTup`.

return: a tuple, every other element of `aTup`.

`polynomials`

Compute the value of a polynomial function at a given value x. Return that value as a float.

inputs: list of numbers (length > 0) and a number (float).

return: float.

A.2.2 User Study on Usefulness

`Fibonacci sequence`

Write a program that takes as input an integer $k > 0$ and prints the integer $n > 0$ such that $F_n \leq k < F_{n+1}$.

Here F_n means the n^{th} number in the Fibonacci sequence defined by the relation:

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \text{ for } n > 2 \\ F_1 &= 1 \\ F_2 &= 1 \end{aligned}$$

Examples of Fibonacci numbers are: $1, 1, 2, 3, 5, 8, \ldots$

`Special number`

Write a program that takes as input an integer $n \geq 0$ and prints `YES` if n is a *special* number, and `NO` otherwise.

A number is special if the sum of cubes of its digits is equal to the number itself.

Note: A cube of some number x is $x^3 = x \cdot x \cdot x$.

For example: 371 is a special number, since $3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$.

`Reverse Difference`

Write a program that takes as input a positive integer $n > 0$ and prints the difference of n and its reverse.

For example: if n is 1234, the output will be -3087 (result of $1234 - 4321$).

`Factorial interval`

Write a program that takes as input two integers n and m (where $0 \leq n \leq m$), and prints the number (*count*) of factorial numbers in the closed interval $[n, m]$.

A number f is a factorial number if there exists some integer $i \geq 0$ such that $f = i!$

Note: $i! = 1 \cdot 2 \cdots i$, that is, $i!$ is a product of first i natural numbers, excluding 0.

Examples of factorial numbers are: $1, 2, 6, 24, 120, \ldots$

`Trapezoid`

Write a program to do the following:

(a) Read height h and base length b as the input.

(b) Print h lines of output such that they form a pattern in the shape of a *regular trapezoid*.

(c) Trapezoid should be formed using the symbol `"*"`.

Example output for $h = 5$ and $b = 14$ (`"-"` denotes where spaces should go, you should print a space `" "` instead of `"-"`):

```
----******
---********
--**********
-************
**************
```

Important: There should be *NO EXTRA SPACE* (before the pattern, between rows, between columns, ...). The last line should be an empty line.

`Rhombus`

Write a program to do the following:

(a) Take height h as the input.

(b) Print h lines of output such that they form a pattern in the shape of a *rhombus*.

(c) Each line should be formed by the integer representing the column number modulo 10.

Note: You can assume that h will be odd and $h \geq 3$.

Example output for $h = 5$ (`"-"` denotes where spaces should go, you should print a space `" "` instead of `"-"`):

```
--3
-234
12345
-234
--3
```

Important: There should be *NO EXTRA SPACE* (before the pattern, between rows, between columns, ...). The last line should be an empty line.

List of Figures

List of Tables

Bibliography

[ADLM15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. "Analysing the Complexity of Functional Programs: Higher-order Meets First-order". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming.* ICFP 2015. Vancouver, BC, Canada: ACM, 2015, pp. 152–164. ISBN: 978-1-4503-3669-7. DOI: `10.1145/2784731.2784753`. URL: `http://doi.acm.org/10.1145/2784731.2784753`.

[AGK13] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. "Automatically Generating Problems and Solutions for Natural Deduction". In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence.* IJCAI '13. Beijing, China: AAAI Press, 2013, pp. 1968–1975. ISBN: 978-1-57735-633-2. URL: `http://dl.acm.org/citation.cfm?id=2540128.2540411`.

[AGP13] Erik Andersen, Sumit Gulwani, and Zoran Popovic. "A Trace-based Framework for Analyzing and Synthesizing Educational Progressions". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI '13. Paris, France: ACM, 2013, pp. 773–782. ISBN: 978-1-4503-1899-0. DOI: `10.1145/2470654.2470764`. URL: `http://doi.acm.org/10.1145/2470654.2470764`.

[AGP14] Eirk Andersen, Sumit Gulwani, and Zoran Popovič. *Programming by demonstration framework applied to procedural math problems.* Tech. rep. 2014.

[AL80] Anne Adam and Jean-Pierre H. Laurent. "LAURA, A System to Debug Student Programs". In: *Artif. Intell.* 15.1-2 (1980).

[Alb+12] Elvira Albert et al. "Cost Analysis of Object-oriented Bytecode Programs". In: *Theor. Comput. Sci.* 413.1 (Jan. 2012), pp. 142–159. ISSN: 0304-3975. DOI: `10.1016/j.tcs.2011.07.009`. URL: `http://dx.doi.org/10.1016/j.tcs.2011.07.009`.

[Ali+10] Christophe Alias et al. "Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs". In: *Static Analysis.* Ed. by Radhia Cousot and Matthieu Martel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 117–133. ISBN: 978-3-642-15769-1.

[Alu+13] Rajeev Alur et al. “Automated Grading of DFA Constructions”. In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI ’13. Beijing, China: AAAI Press, 2013, pp. 1976–1982. ISBN: 978-1-57735-633-2. URL: `http://dl.acm.org/citation.cfm?id=2540128.2540412`.

[Arc08] Andrea Arcuri. “On the Automation of Fixing Software Bugs”. In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion ’08. Leipzig, Germany: ACM, 2008, pp. 1003–1006. ISBN: 978-1-60558-079-1. DOI: `10.1145/1370175.1370223`. URL: `http://doi.acm.org/10.1145/1370175.1370223`.

[BBL98] Prosenjit Bose, Jonathan F. Buss, and Anna Lubiw. “Pattern Matching for Permutations”. In: *Inf. Process. Lett.* 65.5 (1998), pp. 277–283.

[Bey+09] D. Beyer et al. “Software model checking via large-block encoding”. In: *2009 Formal Methods in Computer-Aided Design*. 2009, pp. 25–32. DOI: `10.1109/FMCAD.2009.5351147`.

[Bro+16] Marc Brockschmidt et al. “Analyzing Runtime and Size Complexity of Integer Programs”. In: *ACM Trans. Program. Lang. Syst.* 38.4 (Aug. 2016), 13:1–13:50. ISSN: 0164-0925. DOI: `10.1145/2866575`. URL: `http://doi.acm.org/10.1145/2866575`.

[Cha+11] Satish Chandra et al. “Angelic Debugging”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. New York, NY, USA: ACM, 2011, pp. 121–130. ISBN: 978-1-4503-0445-0. DOI: `10.1145/1985793.1985811`. URL: `http://doi.acm.org/10.1145/1985793.1985811`.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. “Compositional Certified Resource Bounds”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’15. Portland, OR, USA: ACM, 2015, pp. 467–478. ISBN: 978-1-4503-3468-6. DOI: `10.1145/2737924.2737955`. URL: `http://doi.acm.org/10.1145/2737924.2737955`.

[D’a+15] Loris D’antoni et al. “How Can Automatic Feedback Help Students Construct Automata?” In: *ACM Trans. Comput.-Hum. Interact.* 22.2 (Mar. 2015), 9:1–9:24. ISSN: 1073-0516. DOI: `10.1145/2723163`. URL: `http://doi.acm.org/10.1145/2723163`.

[Das+16] Rajdeep Das et al. “Prutor: A System for Tutoring CS1 and Collecting Student Programs for Analysis”. In: *CoRR* abs/1608.03828 (2016). URL: `http://arxiv.org/abs/1608.03828`.

[DLH90] Saumya K. Debray, Nai-Wei Lin, and Manuel Hermnegildo. “Task Granularity Analysis in Logic Programs”. In: *SIGPLAN Not.* 25.6 (June 1990), pp. 174–188. ISSN: 0362-1340. DOI: `10.1145/93548.93564`. URL: `http://doi.acm.org/10.1145/93548.93564`.

[Dru+14] A. Drummond et al. “Learning to Grade Student Programs in a Massive Open Online Course”. In: *Data Mining (ICDM), 2014 IEEE International Conference on.* 2014, pp. 785–790. DOI: 10.1109/ICDM.2014.142.

[DSS16] Loris D’Antoni, Roopsha Samanta, and Rishabh Singh. “Qlose: Program Repair with Quantitative Objectives”. In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II.* 2016, pp. 383–401. URL: http://dx.doi.org/10.1007/978-3-319-41540-6_21.

[DVZ13] Yulia Demyanova, Helmut Veith, and Florian Zuleger. “On the concept of variable roles and its use in software analysis”. In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013.* 2013, pp. 226–230. URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679414.

[DW10] V. Debroy and W.E. Wong. “Using Mutation to Automatically Suggest Fixes for Faulty Programs”. In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on.* 2010, pp. 65–74. DOI: 10.1109/ICST.2010.66.

[FMH14] Antonio Flores-Montoya and Reiner Hähnle. “Resource Analysis of Complex Programs with Cost Equations”. In: *Programming Languages and Systems.* Ed. by Jacques Garrigue. Cham: Springer International Publishing, 2014, pp. 275–295. ISBN: 978-3-319-12736-1.

[For+09] Stephanie Forrest et al. “A Genetic Programming Approach to Automated Software Repair”. In: *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation.* GECCO ’09. Montreal, Qu&#233;bec, Canada: ACM, 2009, pp. 947–954. ISBN: 978-1-60558-325-9. DOI: 10.1145/1569901.1570031. URL: http://doi.acm.org/10.1145/1569901.1570031.

[GAW07] Simon Goldsmith, Alex Aiken, and Daniel Shawcross Wilkerson. “Measuring empirical computational complexity”. In: *ESEC/SIGSOFT FSE.* 2007.

[GKT11] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. “Synthesizing Geometry Constructions”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI ’11. San Jose, California, USA: ACM, 2011, pp. 50–61. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993505. URL: http://doi.acm.org/10.1145/1993498.1993505.

[Gla+14] Elena L. Glassman et al. “OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale”. In: *Proceedings of the Adjunct Publication of the 27th Annual ACM Symposium on User Interface Software and Technology.* UIST’14 Adjunct. Honolulu, Hawaii, USA: ACM, 2014, pp. 129–130. ISBN: 978-1-4503-3068-8. DOI: 10.1145/2658779.2658809. URL: http://doi.acm.org/10.1145/2658779.2658809.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. "SPEED: precise and efficient static estimation of program computational complexity". In: *POPL*. 2009, pp. 127–139.

[GMK11] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. "Specification-based Program Repair Using SAT". In: *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software*. TACAS'11/ETAPS'11. Saarbr&#252;cken, Germany: Springer-Verlag, 2011, pp. 173–188. ISBN: 978-3-642-19834-2. URL: `http://dl.acm.org/citation.cfm?id=1987389.1987408`.

[Gou+15] C. Le Goues et al. "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs". In: *IEEE Transactions on Software Engineering* 41.12 (2015), pp. 1236–1256. ISSN: 0098-5589. DOI: `10.1109/TSE.2015.2454513`.

[GRZ14] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. "Feedback Generation for Performance Problems in Introductory Programming Assignments". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 41–51. ISBN: 978-1-4503-3056-5. DOI: `10.1145/2635868.2635912`. URL: `http://doi.acm.org/10.1145/2635868.2635912`.

[GRZ18] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. "Automated Clustering and Program Repair for Introductory Programming Assignments". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 465–480. ISBN: 978-1-4503-5698-5. DOI: `10.1145/3192366.3192387`. URL: `http://doi.acm.org/10.1145/3192366.3192387`.

[Gul14] Sumit Gulwani. "Example-Based Learning in Computer-Aided STEM Education". In: *To appear in Commun. ACM* (2014). URL: `http://research.microsoft.com/en-us/um/people/sumitg/pubs/education13.pdf`.

[Gup+08] Ashutosh Gupta et al. "Proving non-termination". In: *POPL*. 2008, pp. 147–158.

[GZ10] Sumit Gulwani and Florian Zuleger. "The reachability-bound problem". In: *PLDI*. 2010, pp. 292–304.

[HAH12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. "Resource Aware ML". In: *Computer Aided Verification*. Ed. by P. Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 781–786. ISBN: 978-3-642-31424-7.

[Hea+17] Andrew Head et al. "Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis". In: *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*. L@S '17. Cambridge, Massachusetts, USA: ACM, 2017, pp. 89–98. ISBN: 978-1-4503-4450-0. DOI: 10.1145/3051457.3051467. URL: http://doi.acm.org/10.1145/3051457.3051467.

[Her+05] Manuel V. Hermenegildo et al. "Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and the Ciao System Preprocessor)". In: *Sci. Comput. Program.* 58.1-2 (Oct. 2005), pp. 115–140. ISSN: 0167-6423. DOI: 10.1016/j.scico.2005.02.006. URL: http://dx.doi.org/10.1016/j.scico.2005.02.006.

[Iha+10] Petri Ihantola et al. "Review of Recent Systems for Automatic Assessment of Programming Assignments". In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling '10. Koli, Finland: ACM, 2010, pp. 86–93. ISBN: 978-1-4503-0520-4. DOI: 10.1145/1930464.1930480. URL: http://doi.acm.org/10.1145/1930464.1930480.

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. "Program Repair As a Game". In: *Proceedings of the 17th International Conference on Computer Aided Verification*. CAV'05. Edinburgh, Scotland, UK: Springer-Verlag, 2005, pp. 226–238. ISBN: 3-540-27231-3, 978-3-540-27231-1. DOI: 10.1007/11513988_23. URL: http://dx.doi.org/10.1007/11513988_23.

[JU97] David Jackson and Michelle Usher. "Grading student programs using ASSYST". In: *SIGCSE*. 1997, pp. 335–339.

[Kal+16] Shalini Kaleeswaran et al. "Semi-supervised Verified Feedback Generation". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 739–750. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950363. URL: http://doi.acm.org/10.1145/2950290.2950363.

[KB11] Robert Könighofer and Roderick Bloem. "Automated Error Localization and Correction for Imperative Programs". In: *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*. FMCAD '11. Austin, Texas: FMCAD Inc, 2011, pp. 91–100. ISBN: 978-0-9835678-1-3. URL: http://dl.acm.org/citation.cfm?id=2157654.2157671.

[Ke+15] Yalin Ke et al. "Repairing Programs with Semantic Code Search (T)". In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 295–306. ISBN: 978-1-5090-0025-8. URL: http://dx.doi.org/10.1109/ASE.2015.60.

[Kim+16] Dohyeong Kim et al. "Apex: Automatic Programming Assignment Error Explanation". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: ACM, 2016, pp. 311–327. ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984031. URL: http://doi.acm.org/10.1145/2983990.2984031.

[KK12] Chinmay Kulkarni and Scott R. Klemmer. *Learning design wisdom by augmenting physical studio critique with online self-assessment*. Tech. rep. 2012.

[Lah+15] Shuvendu Lahiri et al. *Automated Differential Program Verification for Approximate Computing*. Tech. rep. 2015. URL: https://www.microsoft.com/en-us/research/publication/automated-differential-program-verification-for-approximate-computing/.

[LR16] Fan Long and Martin Rinard. "Automatic Patch Generation by Learning Correct Code". In: *SIGPLAN Not.* 51.1 (Jan. 2016), pp. 298–312. ISSN: 0362-1340. URL: http://doi.acm.org/10.1145/2914770.2837617.

[Mas11] Ken Masters. "A Brief Guide To Understanding MOOCs". In: *The Internet Journal of Medical Education* 1.2 (2011). DOI: 10.5580/1f21. (Visited on 09/24/2012).

[Mil71] Robin Milner. *An Algebraic Definition of Simulation Between Programs*. Tech. rep. Stanford, CA, USA, 1971.

[MKK17] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. "Contract-based Resource Verification for Higher-order Functions with Memoization". In: *SIGPLAN Not.* 52.1 (Jan. 2017), pp. 330–343. ISSN: 0362-1340. DOI: 10.1145/3093333.3009874. URL: http://doi.acm.org/10.1145/3093333.3009874.

[Mur87] William R. Murray. "Automatic program debugging for intelligent tutoring systems". In: *Computational Intelligence* 3 (1987).

[Nec00] George C. Necula. "Translation Validation for an Optimizing Compiler". In: *SIGPLAN Not.* 35.5 (May 2000), pp. 83–94. ISSN: 0362-1340. DOI: 10.1145/358438.349314. URL: http://doi.acm.org/10.1145/358438.349314.

[Ngu+14] Andy Nguyen et al. "Codewebs: Scalable Homework Search for Massive Open Online Programming Courses". In: *Proceedings of the 23rd International Conference on World Wide Web*. WWW '14. Seoul, Korea: ACM, 2014, pp. 491–502. ISBN: 978-1-4503-2744-2. DOI: 10.1145/2566486.2568023. URL: http://doi.acm.org/10.1145/2566486.2568023.

[Nis+13] Adrian Nistor et al. “Toddler: Detecting Performance Problems via Similar Memory-access Patterns”. In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE ’13. San Francisco, CA, USA: IEEE Press, 2013, pp. 562–571. ISBN: 978-1-4673-3076-3. URL: `http://dl.acm.org/citation.cfm?id=2486788.2486862`.

[NX13] Khanh Nguyen and Guoqing Xu. “Cachetor: Detecting Cacheable Data to Remove Bloat”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: ACM, 2013, pp. 268–278. ISBN: 978-1-4503-2237-9. DOI: `10.1145/2491411.2491416`. URL: `http://doi.acm.org/10.1145/2491411.2491416`.

[PSS98] A. Pnueli, M. Siegel, and F. Singerman. “Translation Validation”. In: Springer, 1998, pp. 151–166.

[Rad+17] Ivan Radiček et al. “Monadic refinements for relational cost analysis”. In: *Proc. ACM Program. Lang.* 2.POPL (2017), 36:1–36:32. ISSN: 2475-1421. DOI: `10.1145/3158124`. URL: `http://doi.acm.org/10.1145/3158124`.

[RK17] Kelly Rivers and Kenneth R. Koedinger. “Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor”. In: *International Journal of Artificial Intelligence in Education* 27.1 (2017), pp. 37–64. ISSN: 1560-4306. DOI: `10.1007/s40593-015-0070-z`. URL: `https://doi.org/10.1007/s40593-015-0070-z`.

[Rol+17] Reudismam Rolim et al. “Learning Syntactic Program Transformations from Examples”. In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE ’17. Buenos Aires, Argentina: IEEE Press, 2017, pp. 404–415. ISBN: 978-1-5386-3868-2. DOI: `10.1109/ICSE.2017.44`. URL: `https://doi.org/10.1109/ICSE.2017.44`.

[SG11] Michael Striewe and Michael Goedicke. “Using run time traces in automated programming tutoring”. In: *ITiCSE*. 2011, pp. 303–307.

[SG13] Michael Striewe and Michael Goedicke. “Trace Alignment for Automated Tutoring”. In: *CAA*. 2013.

[SGR12] Rohit Singh, Sumit Gulwani, and Sriram Rajamani. “Automatically Generating Algebra Problems”. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI’12. Toronto, Ontario, Canada: AAAI Press, 2012, pp. 1620–1627. URL: `http://dl.acm.org/citation.cfm?id=2900929.2900958`.

[SGSL13] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. “Automated feedback generation for introductory programming assignments”. In: *PLDI*. 2013, pp. 15–26.

[SJB05] Stefan Staber, Barbara Jobstmann, and Roderick Bloem. “Finding and Fixing Faults”. English. In: *Correct Hardware Design and Verification Methods.* Ed. by Dominique Borrione and Wolfgang Paul. Vol. 3725. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 35–49. ISBN: 978-3-540-29105-3. DOI: 10.1007/11560548_6. URL: http://dx.doi.org/10.1007/11560548_6.

[SMK01] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. “Fully Automatic Assessment of Programming Exercises”. In: *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education.* ITiCSE ’01. Canterbury, United Kingdom: ACM, 2001, pp. 133–136. ISBN: 1-58113-330-8. DOI: 10.1145/377435.377666. URL: http://doi.acm.org/10.1145/377435.377666.

[Suz+17] Ryo Suzuki et al. “Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments”. In: *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems.* CHI EA ’17. Denver, Colorado, USA: ACM, 2017, pp. 2951–2958. ISBN: 978-1-4503-4656-6. DOI: 10.1145/3027063.3053187. URL: http://doi.acm.org/10.1145/3027063.3053187.

[SZV14] Moritz Sinn, Florian Zuleger, and Helmut Veith. “A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis”. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings.* 2014, pp. 745–761. DOI: 10.1007/978-3-319-08867-9_50. URL: http://dx.doi.org/10.1007/978-3-319-08867-9_50.

[SZV17] Moritz Sinn, Florian Zuleger, and Helmut Veith. “Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints”. In: *Journal of Automated Reasoning* 59.1 (2017), pp. 3–45. ISSN: 1573-0670. DOI: 10.1007/s10817-016-9402-4. URL: https://doi.org/10.1007/s10817-016-9402-4.

[Tai79] Kuo-Chung Tai. “The Tree-to-Tree Correction Problem”. In: *J. ACM* 26.3 (July 1979), pp. 422–433. ISSN: 0004-5411. DOI: 10.1145/322139.322143. URL: http://doi.acm.org/10.1145/322139.322143.

[Til+13a] Nikolai Tillmann et al. “Teaching and Learning Programming and Software Engineering via Interactive Gaming”. In: *Proc. 35th International Conference on Software Engineering (ICSE 2013), Software Engineering Education (SEE).* San Francisco, CA, 2013. URL: http://www.cs.illinois.edu/homes/taoxie/publications/icse13see-pex4fun.pdf.

[Til+13b] Nikolai Tillmann et al. “Teaching and learning programming and software engineering via interactive gaming”. In: *ICSE.* 2013.

[Til+14] Nikolai Tillmann et al. “Code Hunt: Searching for Secret Code for Fun”. In: *Proceedings of the International Conference on Software Engineering (Workshops)* (2014). URL: `http://research.microsoft.com/apps/pubs/default.aspx?id=210651`.

[Uno97] Takeaki Uno. “Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs”. In: *ISAAC*. 1997, pp. 92–101.

[Wel+12] Daniel S. Weld et al. “Personalized online education - a crowdsourcing challenge”. In: *In Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*. 2012.

[WSS18] Ke Wang, Rishabh Singh, and Zhendong Su. “Search, Align, and Repair: Data-driven Feedback Generation for Introductory Programming Exercises”. In: *SIGPLAN Not.* 53.4 (June 2018), pp. 481–495. ISSN: 0362-1340. DOI: `10.1145/3296979.3192384`. URL: `http://doi.acm.org/10.1145/3296979.3192384`.

[Yi+17] Jooyong Yi et al. “A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 740–751. ISBN: 978-1-4503-5105-8. DOI: `10.1145/3106237.3106262`. URL: `http://doi.acm.org/10.1145/3106237.3106262`.

[ZS89] K. Zhang and D. Shasha. “Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems”. In: *SIAM J. Comput.* 18.6 (Dec. 1989), pp. 1245–1262. ISSN: 0097-5397. DOI: `10.1137/0218082`. URL: `http://dx.doi.org/10.1137/0218082`.