CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Javista

Automated Assessment of Imperative Programs

Bachelor of Science Thesis in Computer Science Engineering

MAXIMILIAN ALGEHED
SIMON BOIJ
MAZDAK FARROKHZAD
JOEL HULTIN
ALEKSANDER STERN KAAR

Chamers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden

Bachelor of Science Thesis

Javista

A tool for Automated Assessment of Programming Exercises

MAXIMILIAN ALGEHED
SIMON BOIJ
MAZDAK FARROKHZAD
JOEL HULTIN
ALEKSANDER STERN KAAR

Department of Computer Science and Engineering
Chamers University of Technology
University of Gothenburg

Göteborg, Sweden, 2017

Javista
A tool for Automated Assessment of Programming Exercises

MAXIMILIAN ALGEHED
SIMON BOIJ
MAZDAK FARROKHZAD
JOEL HULTIN
ALEKSANDER STERN KAAR

Examiner: Niklas Broberg
Supervisor: Alex Gerdes

Department of Computer Science and Engineering
Chamers University of Technology
University of Gothenburg
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg 2017

Javista

A tool for Automated Assessment of Programming Exercises

MAXIMILIAN ALGEHED
SIMON BOIJ
MAZDAK FARROKHZAD
JOEL HULTIN
ALEKSANDER STERN KAAR

Department of Computer Science and Engineering
Chamers University of Technology
University of Gothenburg

Bachelor of Science Thesis

Abstract

This thesis presents a methodology and a tool for automated assessment of programming exercises, with the purpose of reducing the workload of teachers. Our aim is for the tool to provide accurate and useful assessment given an exercise specification. Using the tool could allow teachers to spend more time helping students. The tool, implemented in Haskell, is intended to be used by teachers through a command line interface and targets a subset of Java. Assessment is achieved by using semantic and behavioural analysis. Semantic analysis consists of normalisation and prefix trees, while behavioural analysis consists of testing including integrated shrinking. The presented tool is evaluated using a data set from the course TDA450 at Chalmers University of Technology. The tool managed to classify 60% of the solutions as either correct or incorrect with no false positives. The result shows that it is possible to automatically assess student solutions and suggests that more solutions can be classified given further development.

Keywords: Automated Assessment, Normalisation, Strategies, Property Based Testing, Programming Language Technology, Java

Sammanfattning

Rapporten presenterar en metod och ett verktyg för automatisk rättning av programmeringsövningar, med syftet att minska lärarnas arbetsbelastning. Vårt mål är att verktyget ska ge en korrekt och användbar bedömning, givet en problembeskrivning. Genom att använda verktyget skulle lärare kunna spendera mer tid med att hjälpa studenter. Verktyget, implementerat i Haskell, är avsett att användas av lärare via ett kommandoradsgränssnitt och hanterar en delmängd av Java. För att rätta används analys av både semantik och beteende. Semantisk analys består av normalisering och prefix-träd, medan beteendeanalys innefattar testning och integrerad krympning. Det presenterade verktyget utvärderas med hjälp av ett dataset från kursen TDA450 vid Chalmers tekniska högskola. Verktyget lyckades klassificera 60% av lösningarna som antingen korrekta eller felaktiga utan felaktiga klassificeringar. Resultatet visar att automatisk rättning är möjlig och indikerar att fler lösningar kan klassificeras givet fortsatt utveckling.

Acknowledgements

We would like to extend a special thanks to our supervisor Alex Gerdes, for giving us the opportunity of working on this bachelor thesis and for all his help.

We would also like to thank Thomas Hallgren, Michal Palka, Christian Persson, Jacob Holmgren, and Rickard Hjort for their constructive feedback in opposition to our thesis.

Contents

1 Introduction

Teaching programming at university level requires much effort to assess the quality of student solutions to programming exercises. This problem has given rise to the research area of automated assessment. Approaches to automatically assessing the quality of student solutions to programming exercises include: testing [1], using Embedded Domain Specific Languages (EDSLs) to express desirable properties of student solutions [2], and using programming strategies [3] to identify student solutions as variants of predefined model solutions [4]. Much work has gone into automated assessment for the functional programming language Haskell using strategies and model solutions [5]. While this work has been successful, attempts to transfer it to object oriented languages have seen limited success [4], focusing on solutions written in the context of a programming tutoring tool rather than grading student submissions to assignments and hand-ins. Programming tutoring tools usually constrain the way students write their programs and offer hints to help students. This means that student programs written in the context of a programming tutor can be expected to vary less than student submissions to programming assignments.

There are many possible ways to reduce the work required to grade student solutions. Some different aspects of grading student solutions may not be simple to automate, such as assessing the relevance of comments or the quality of variable names. While this is an interesting area of research, this report is not concerned with such assessment. Instead this report focuses on assertions about program correctness in the context of students solutions to well defined programming exercises in early university level programming courses on imperative and object oriented programming. While automatically assessing the correctness of arbitrary computer programs is a difficult problem, the limited domain of assessing programming exercises greatly reduces the complexity. In the context of automated assessment, specifications can be made very detailed and correct reference, or model, programs generally exist.

1.1 Purpose

The aim of this project is to create a tool for objective and consistent automated assessment of programming exercises written in a subset of Java [6]. The tool is meant as an aid in assessing solutions in order to reduce the workload for the teacher. The tool initially targets an introductory course in imperative and object oriented programming at Chalmers university of technology [7] to provide teachers with assistance in assessing student solutions to programming exercises. In order for the tool to provide relevant assessment there needs to be a way for a teacher to provide it with an exercise specification. The goal is to provide accurate and useful assessment with as simple an exercise specification as possible. While initially targeted at an introductory course, the tool should be sufficiently extensible that it may, in the future, be used in more advanced courses also taught in Java.

1.2 Problem Description

The main problem is to classify student solutions to programming exercises as either correct or incorrect. If the tool classifies a solution as correct that should be an indication that the solution should be accepted without further inspection, if the solution is classified as incorrect it should be rejected. The tool must also be able to indicate when the student solution is unclassifiable in the case where the student solution can not be classified as either correct or incorrect.

The tool needs to accept an exercise specification from a teacher, called model solution, which is used to assess student solutions as either correct, incorrect, or unclassifiable. Such a specification should include one or more model solutions as well as a specification of valid program input formats.

An important sub-problem addressed in this report is to find a useful definition of correctness. Such a definition must express what it mean for a student solution to be correct with respect to a programming exercise specification. The notion of correctness should be irrespective of syntactic

and stylistic choices on behalf of the student. Examples of such stylistic choices include using `for`-loops instead of `while`-loops, ordering independent program statements in different ways, using different variables names etc. Similarly, a useful definition of incorrectness is necessary.

The final problem addressed in this report is to combine the notions of correctness, incorrectness, and unclassifiability to implement a tool for automated assessment. It is important that the tool is reliable, never assessing incorrect solutions as correct and vice verse. However, it is also important that the tool is useful, a tool which always assesses a student solution as unclassifiable is not a useful tool.

1.3 Scope

While the tool presented in this report is aimed at the Java programming language, it does not support the entire language. The focus is on a small subset of the language, including:

- Conditional statements
- Primitive types and the `String` class
- Arrays
- Loops
- Input and output
- Static methods
- Object instantiation and object types

The tool provides a command line interface (CLI) for assisting teachers and providing them with automated assessment of student programs. The functionality of the tool is limited in the following way:

- The tool does not provide an interface for students.
- The tool does not provide a Graphical User Interface (GUI).
- The tool does not provide a way of assessing incomplete student solutions. As such the tool does not act as a programming tutor.

In conclusion, the scope is limited to an assessment tool, for use by teachers, and targets a subset of the Java programming language.

1.4 Contributions

We present a methodology and a tool for objective automated assessment of programming exercises in a course for basic imperative programming using a subset of the Java [6] programming language. By providing such assessments, the workload for teaching assistants could be reduced and shifted from manual assessment into one-time setup costs for a specific exercise. This cost is to provide model solutions [4] to programming exercises and input format specifications. This leaves teachers with more time to assist and teach students.

This report presents the following contributions

- A conceptual architecture for automated assessment of student solutions to programming exercises, presented in Section 3. This section presents solutions to general problems with automatic assessment as well as more specific problems for the chosen domain.
- Several Haskell EDSLs, presented in Section 4, which implement the individual conceptual solutions from Section 3.

- A tool, written in Haskell, for automated assessment of Java programming exercises based on the architecture presented in Section 3. The implementation of the tool is presented in Section 4 and example usage is presented in Section 5.

2 Theoretical Background

This section presents the theoretical background necessary to understand the rest of the report. The reader is assumed to be familiar with basic concepts from programming language theory and practise as well as the Java programming language.

Model Solution A pre-defined solution to a programming exercise provided by a teacher. A model solution is a guaranteed correct solution to an exercise.

Embedded Domain Specific Language A domain specific language (DSL) is a language specialised for some particular purpose. Writing a specialised DSL has the advantages of making it easier to create solutions for some specific problem. Furthermore, an Embedded Domain Specific Language (EDSL) is a DSL which is *embedded* in some host language, EDSLs are very similar to libraries but generally provide an entire programming model rather than a set of library functions [8].

Abstract Syntax Tree An abstract syntax tree (AST) is a tree representation of a term, such as a mathematical expression, in some language. ASTs are data structures which formal-language based tools such as compilers can work with. They are initially produced by using a parser on source code the programmer has written. After parsing a `.java` file to an AST, syntactic differences such as white-space becomes insignificant [9].

2.1 Equivalence

There are many different ways of defining equivalence. This section presents some of these ways.

Syntactic Equivalence Comparing two programs syntactically can be done once they are parsed into ASTs. If their respective ASTs are equivalent after having been parsed with the same parser, then the programs are considered syntactically equivalent.

Semantic Equivalence Semantic equivalence is when two programs have the same meaning. This implies that two solutions might be written in different ways, although they describe the same procedure. This means that for all inputs, the semantically equivalent programs would produce the same output.

Behavioural Equivalence Behavioural equivalence [10], [11] is when two programs have the same behaviour. The same behaviour means that two programs produce the same output.

2.2 Static Analysis

Static analysis is analysis that is performed on program text. This means that the analysis is not performed during runtime [12]. Static analysis can be performed to determine the syntactic equivalence of two programs, which implies semantic equivalence. Thus, there is no input space to search. If static analysis determines that a solution is recognised to be equivalent to a model

solution, it provides certainty that the solution is correct. When recognition fails, it is not possible to state that the student solution is incorrect. In this case, correctness is unknown.

Fixed Point A value x is a fixed point of a function f if and only if $f(x) = x$ [13]. This means that applying a function to one of its fixed points returns the same value. Not all functions have fix-points, for example $f(x) = x - 1$. Some functions have more than one, for example $f(x) = |x|$, which computes the absolute value, has all positive numbers and zero as its fixed points.

(Unique) Normal Form All terms that are semantically equivalent have the same normal form. This normal form represents a standardised way to express all semantically equivalent terms [4]. A normal form is defined according to the semantics in a given context. If there only exists one normal form for a term, it is called an unique normal form. An example of a normal form is disjunctive normal form (DNF) in first order logic (FOL). All such expressions have a unique normal form and two expressions with the same truth tables, that is to say the same semantics, have the same DNFs.

Normalisation Normalisation rules are functions that transform semantically equivalent terms into syntactically equivalent terms [14]. The core idea of normalisation is that semantically equivalent terms, for some definition of equivalent, have the same normal form [4]. Thus normalisation must preserve semantics. A term in some language is in normal form when a fix-point has been reached [14].

Consider the process of eliminating double negation from an expression in FOL, given $\neg\neg p$, it can be transformed to p [15]. The expression can not be transformed any further and is thus in normal form. In this paper, this normalisation rule is denoted as, $\neg\neg p \Downarrow p$, which reads as: *"$\neg\neg p$ rewrites to p"*.

2.3 Behavioural Testing

Behavioural testing is the process of testing if two programs have the same behaviour during runtime. This is usually done by checking if, given a command, a program displays the correct behaviour. While this method may provide some certainty that a program is correct, though it can not prove correctness. Some bugs can be difficult to find and arise only for very specific inputs. Assuming deterministic behaviour, unless the entirety of the input space is tested, it is impossible to state that a programs is correct. Since the input space is likely to be large it is unlikely that such evidence can be found in a practical amount of time [16]. Therefore, testing may at best give concrete evidence that a program is incorrect [17].

Property A property, P, of a program f is a judgement on the form of *given some input X, the statement $P(f, X)$ is true* [18]. As an example, consider a program which, given a list of numbers, produces the same list of numbers in a sorted order. A property of this program is that the output of the program given *any* list, is ordered in ascending order. That is, $P(sort, X) = ordered(sortX)$.

Test Case Testing is the act of checking if a given property holds for some input. Each set of input for testing a property is called a test case. In the case that a test case fails, a program is guaranteed to be incorrect.

QuickCheck The Haskell library QuickCheck [18] can generate random input and use it to test if a property holds. It provides functionality for specification of inputs and properties. Some common functions for specifying the generation of inputs are:

- `arbitrary` which generates any value of a specific type.

- `suchThat` which specifies constraints for the generated value.
- `listOf` which generates a list of a specific type.

Generating a list of even `Int` is illustrated by the example shown in Snippet 2.1.

```
listOfEven :: Gen [Int]
listOfEven = listOf $ arbitrary `suchThat` even
```

Snippet 2.1: Function for generating a list of even `Int`.

QuickCheck Generator Input values can be generated by using a QuickCheck Generator. The generator is used by calling the function `generate` which returns a random value with a specified type. Random values can be large if there are no upper or lower bound specified in the generation. A way to make the value smaller is called shrinking. Shrinking is only done when a test case fails for a specific input. That input is then used to find a smaller test case that fails. An example of this is a list of length 10 fails a test case, by splitting the list in two the input is smaller and can be tested again. If it fails, the list can be split again to see if a smaller list fails, if not the last test case that fails is the 1-minimal failing test case. In QuickCheck every type has an associated `shrink` function. This means that every type shrinks in different ways.

1-minimal failing test A 1-minimal failing test [19] is when the input does not fail if the input is shrunk. For example, an array the 1-minimal failing test is; any one element is removed from the array and the test passes, then the previous input is the 1-minimal failing test, thus every element matters.

Integrated Shrinking A method, used in the `disorder-jack` tool [20] among others, is to integrate the shrinking with the generation of test data. Integrating the shrinking with the generation work by generating the test case as well as the ways of shrinking that test case at the same time. This is in contrast to the method used in QuickCheck, where shrinking is independent of the generator in use.

3 Conceptual Solutions

This section gives a conceptual overview of how normalisation, matching and testing are used in this project to automatically assess student solutions. It also gives a definition of correctness, incorrectness, and unclassifiability.

If a student solution is semantically equivalent to a correct model solution, the student solution is correct since they have the same meaning. Static analysis can be performed to determine the syntactic equivalence of two programs, which implies semantic equivalence. Correctness of a student solution in this context is therefore defined as semantic equivalence to a model solution. This guarantees that a student solution is correct if the static analysis, used to determine semantically equivalence, deems two solutions to be equal. When a solution can not be recognised to be equal, correctness is unknown.

A student solution that has a different behaviour than all model solutions is incorrect. This can be found using behavioural testing. Incorrectness or in-equivalence in this context means that a student solution does not behave the same way as a model solution. Testing the behavioural equivalence is used to determine the inequality of two solutions which guarantees that a student solution is incorrect. If this method fails to determine incorrectness it may at best give an indication that a student solution and a model solution is equal.

Unclassifiable may also be a result of the combination of these two methods. At least, if many different tests are done, the indication that the student solution is correct may be stronger and the

result is not entirely unknown. The process of recognising student solutions as correct or incorrect is divided in to stages as shown in Figure 3.1. First the student solution and all model solutions are parsed to ASTs. Then semantic analysis is done with the two methods, normalisation and matching, if semantic equality can not be guaranteed, testing is used to determine behavioural in-equality. Finally, information about the solution is provided to the teacher using the tool.

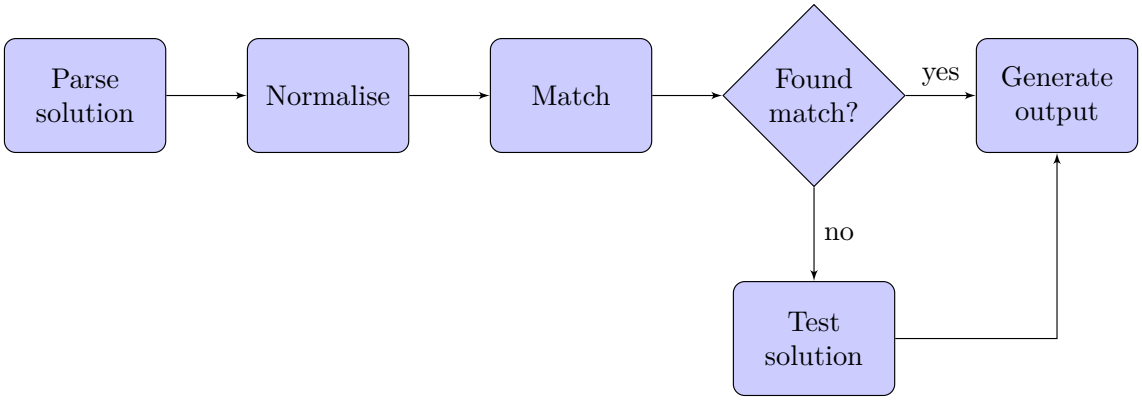

Figure 3.1: An outline of the pipeline.

3.1 Normalisation

Consider an assignment where a student has to solve the problem of summing up all numbers from 1 to n, where $n \in \mathbb{N}$. This can be solved in many different ways. One way is to implement a `for` loop going from i to n, using i to add to a sum variable. However if a student were to use a `while` loop instead of a `for` loop, then the ASTs would not be syntactically equivalent, even though the semantics would be the same.

Student

```
int sum = 0;
for (int i = 1; i ⩽ n; i++)
    sum = sum + i;
```

Model

```
int sum = 0;
int i = 1;
while (i ⩽ n) {
    sum = sum + i;
    i++;
}
```

Snippet 3.1: A student solution containing a `for` loop and its equivalent model solution containing a `while` loop.

In this case, normalisation can be used to minimise the amount of variations. Thus, equivalence is redefined such that normalisation is applied to both ASTs before doing a strict identity check. As a consequence, more solutions will be recognised which can be seen in Snippet 3.2.

```
equiv :: AST → AST → Bool
equiv l r = normalise l == normalise r
```

Snippet 3.2: Function for equivalence defined by applying normalisation first

To provide a certainty of correctness, preserving the semantics is also important for automated assessment. The semantics of Java are defined by the Java Language Specification (JLS) [21]. Thus, normalisation rules must adhere to the JLS in order to be semantic preserving.

3.1.1 Normalisation EDSL

An imperative language such as Java has many complicated constructs and is a language in which it is possible to write the same thing in many different ways [22]. This requires writing a large amount of normalisation rules to recognise all ways.

To this end the normalisation EDSL, used while constructing rules, should make it easy to write rules which satisfy the following properties:

- Simplicity: Rules should be simple, allowing them to be easily tested and maintained.
- Composability: Rules must compose well with others, making it possible to construct complicated rules from simpler ones.
- Performance: Rules must be fast to execute since there are many of them and they are executed many times.
- Usability: It must be easy to define rules. For terms consisting of sub-terms, normalisation must have the property that: *a change in any sub-term implies a change in the parent*. By induction, *a change in any term at any depth implies a change in the root term*. Manually tracking whether change occurred in any branch or sub-term makes normalisers hard to write.

In order to recognise a student solution as equivalent to a model solution, their respective variable names must be the same. Consider the task of adding two variables encoded in the solutions given in Snippet 3.3. These solutions are semantically equivalent, up to the variable names, causing recognition to fail. The example in Snippet 3.3 is also one of shadowing, wherein variables in Java may be in different scopes and still have the same name.

Student

```
{
    int y = 0;
    int x = 1;
}
int x = 2;
```

Model

```
{
    int right = 0;
    int sum = 1;
}
int left = 2;
```

Snippet 3.3: A student solution and a model solution with varying variable names.

To remedy these issues, variable names are renamed with new and unique names. This process is called α-renaming [23] and can be used as a normalisation rule. After applying α-renaming to the solutions in Snippet 3.3, they will, look as in Snippet 3.4.

Student

```
{
    int var1 = 0;
    int var2 = 1;
}
int var3 = 2;
```

Model

```
{
    int var1 = 0;
    int var2 = 1;
}
int var3 = 2;
```

Snippet 3.4: Syntactic equivalence of a student solution and a model solution after α-renaming.

3.1.2 Ordering of Composition and Execution

Many normalisation rules depend on variables having unique and predictable names, and having no shadowing occur. By definition, unique variable names implies that there is no shadowing. An example of such a rule is one which first splits declaration and initialisation and then moves all declarations to the top. If this rule is applied before α-renaming is done, the student solution in Snippet 3.3 will look as in Snippet 3.5.

```
int y; y = 0;
int x; x = 1;
int x; x = 2;
{}
```

Snippet 3.5: Moving variables to top before α-renaming causes a scoping error due to redeclaring `x`.

Since normalisation must be semantic preserving, type correctness must also be preserved, which the normalised snippet in Snippet 3.5 does not. Applying α-renaming is therefore crucial for ensuring that rules are truly semantic preserving. Assuming an AST has been α-renamed is also reasonable because the rule would have to check if the AST has shadowing otherwise.

Now consider the following rules, running in the following order:

1. *alpha.var* - which α-renames variables,
2. *do.to.while* - which transforms do-while loops into while loops,
3. *decl.top* - which moves declarations to the top,

on the statement `do { int x; } while (true);`. The statement is rewritten as follows:

start

```
do {
  int x;
}
while (true);
```

⇒

alpha.var

```
do {
  int v1;
}
while (true);
```

⇒

do.to.while

```
{
  int v1;
}
while (true) {
  int v1;
}
```

⇒

decl.top

```
int v1;
int v1; // SCOPING ERROR
{}
while (true) {}
```

Snippet 3.6: Normalisation breaks the Java scoping rules by declaring two variables with the same name.

The examples Snippet 3.5 and Snippet 3.6 demonstrate that the order in which normalisations are run matter. They can not be run in an arbitrary order since it might result in a non semantic preserving . A method of defining dependencies, or at least an order between normalisations is therefore necessary. In this case, *alpha.var* must be applied again before applying *decl.top* to make the reduction semantic preserving.

3.2 Matching

Finding a unique normal form for all equivalent programs is, if at all possible, very difficult. Particularly, when two statements are independent of each other, reordering them with respect to each other does not change the semantics of the program. It is infeasible to define a total order on the statements of a program which guarantees that two semantically equivalent programs are ordered the same way. Therefore, to correctly recognise all variants of a correct solution they need to matched against every valid reordering of that solution. However, even just the three statements in Snippet 3.7 can be reordered in three different ways without affecting the semantics. In the worst case, the number of permutations grows factorially with respect to the number of statements.

Model solution

```
int j;
i = 0;
j = 1;
```

Alt. solution 1

```
int j;
j = 1;
i = 0;
```

Alt. solution 2

```
i = 0;
int j
j = 1;
```

Snippet 3.7: Permutations of a model solution.

The high number of possible permutations makes it impractical to create individual model solutions for each permutation. All semantically equivalent permutations of a model solution should instead

be generated from that model solution. This creates the problem of recognising in which ways it is possible to reorder statements without changing the semantics of the program.

Keuning et. al. [24] describes four scenarios in which a statement a may depend on a previous statement b. These rules are all demonstrated in Snippet 3.8.

- If a uses a variable that is changed in b, then a is dependent on b.
- If a changes or uses a variable which is changed in b, then a is dependent on b.
- No statement can be guaranteed to be independent of a statement for which it is impossible to identify its side-effects, a so called impure statement. If b is impure, a is dependent on b.
- It is impossible to change the placement of a statement which dictates if successive statements are executed or not. If b is such a statement, a is dependent on b.

A)
```
x = x + 1;
int y = x;
```

B)
```
int y = x;
x = x + 1;
```

C)
```
int y = impure();
x = x + 1;
```

D)
```
break;
x = x + 1;
```

Snippet 3.8: Examples where the second statement depends on the first.

3.2.1 Prefix Trees

In order to efficiently compare a student solution to all permutations of a model solution without needlessly generating a possibly large number of permutations we create a data structure we call a prefix tree. A prefix tree is a tree where each edge represents one step in the generation of a complete solution. Each node represents a state of the solution and contains either a complete solution or a prefix with one or more holes. A hole represents a part of a solution that still needs to be defined or expanded upon in order to reach a finished solution. A prefix tree is a representation of the steps required to create a correct solution. An example of a model solution and its corresponding prefix tree is shown in Figure 3.2. While the use of holes to incrementally refine programs is inspired by Gerdes et al. [14], the prefix-tree construction is novel.

Model Solution
```
int j;
i = 0;
j = 1;
```

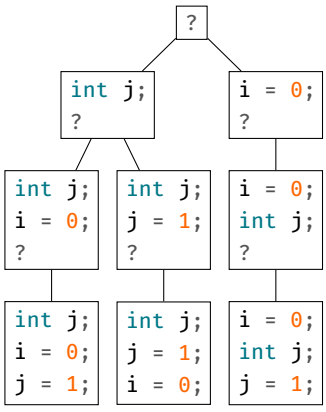

Figure 3.2: A model solution and its prefix tree.

The construction of a prefix tree begins with the root. The root contains only a hole, denoted by a question mark. The children will be the possible states of the solution after one step of generation is done. This generation is done by replacing a hole with a single step required to reach the solution, as well as new holes representing parts of the solution that still need to be added. The process is repeated for each successive node, making each level in the tree another step towards a final solution. When a node is created that has no holes, which means there is no way to extend it

further, it will be a leaf of the tree and represent a finished permutation of the original solution. Each leaf in the tree will contain a permutation of the original solution.

3.2.2 Matching Using Prefix Trees

Simply generating all semantically equivalent permutations of a model solution and checking if the student solution is equal to any of them is computationally unfeasible. Therefore a faster solution is needed. Matching the student solution to the right in Snippet 3.9 with the model solution to the left requires comparing the student solution to the three different permutations of the model solution.

Model

```
int j;
i = 0;
j = 1;
```

Student

```
i = 0;
int j;
j = 1;
```

Snippet 3.9: Semantically equivalent student and model solution.

It is possible to instead match the solution while generating the prefix tree. This matching can be done during the generation of the tree, by checking if the student solution is still possible to achieve from any given node. The matching of the solutions in Snippet 3.9 would start by generating the left child of the root as shown in Figure 3.3. Since the node does not match any prefix of the student solution, the traversal will not continue through that node. In the next step the matching process will instead generate the right node and continue the matching from there. The matching succeeds if it reaches a leaf that is equivalent to the student solution. If it never does, the matching fails. In Figure 3.3 the example succeeds once it reaches the rightmost leaf.

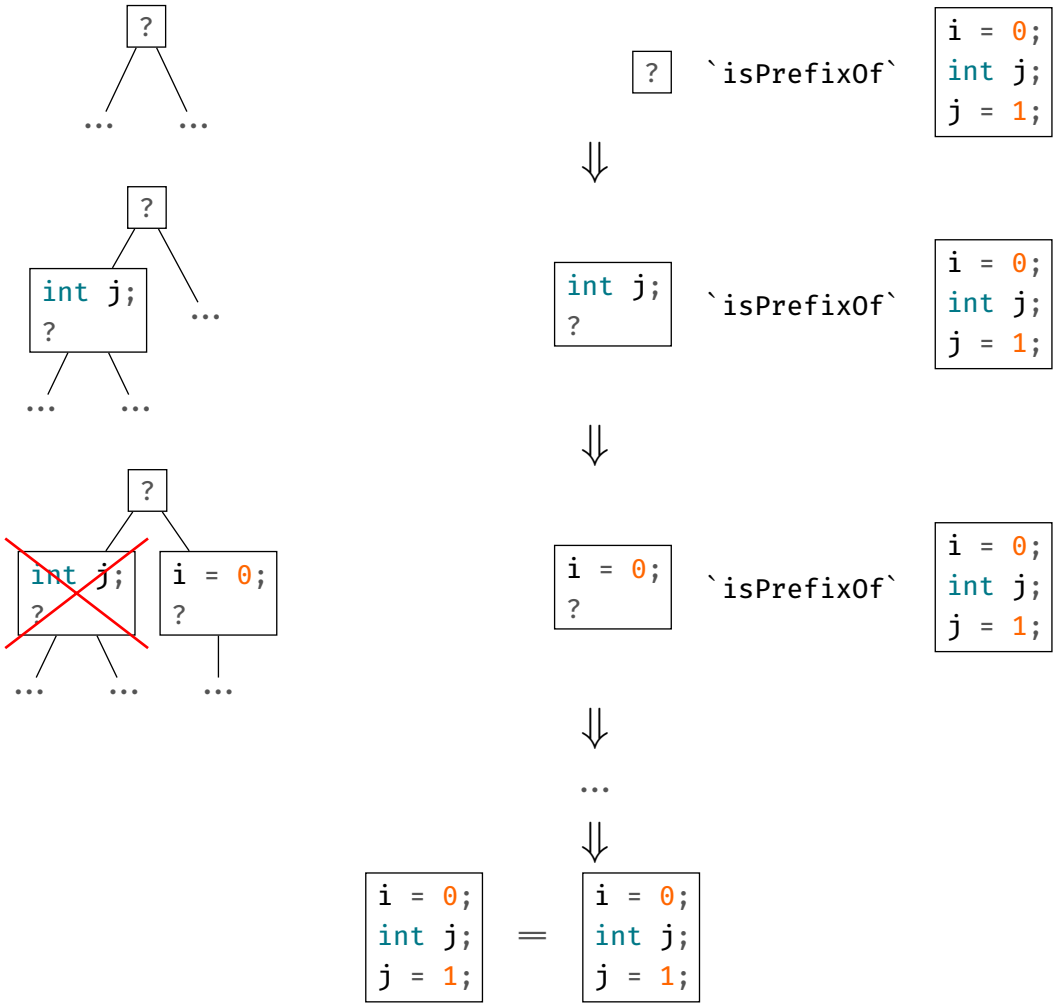

Figure 3.3: The process of matching during tree generation using depth first search and a function `isPrefixOf` *which determines if a partial program is a prefix of a complete program.*

In the worst case, this method of matching will still require traversal through each leaf before it is known whether or not the two solutions match. This means that this method, in the worst case, needs to check solutions for equality $O(n!)$ times. The big gain of the method is the ability to ignore paths in the tree, which is done in two main ways. The first is the aforementioned discarding of paths where the prefix does not match the student solution. Consider for example the two programs in Snippet 3.10. While there exists multiple permutations of the model solution matching will fail immediately when comparing `if` and `while`, without generating any of the permutations. The other way to discard paths is to terminate the traversal upon reaching a matching leaf, making a depth first approach appropriate.

Model

```
if (true) {
    int j;
    i = 0;
    j = 1;
}
```

Student

```
while (true) {
    int j;
    i = 0;
    j = 1;
}
```

Snippet 3.10: A student and a model solution that are not equivalent.

It is, in theory, possible to solve other identification-problems using prefix trees, by from each

node generating each possible node that have the same normal form as that node. This mean that given an infinite amount of memory, computing power or time, there would be no need for normalisation at all. In contrast to normalisation rules which reduces the search space this would instead make the number of possible nodes substantially higher. The matching should therefore not be used separately, but the solutions wanted to be matched should first be normalised. Since the prefix-tree is used to create solutions that have different unique normal form is it sometimes required to renormalise each term that is to be checked for equivalence.

3.3 Testing

Testing is used to determine the incorrectness of a student solution and provide a good indication that it is correct without guaranteeing it. Consider the example exercise specification: read a number n from `stdin` then read n numbers and print their sum. A student and model solution pair for this exercise can be seen in Snippet 3.11. If all the tests pass it can at best give an indication that the solution is correct, but if a failing test case is found the student solution is guaranteed to be incorrect.

Student

```java
Scanner sc =
    new Scanner(System.in);
int x = sc.nextInt();
int sum;
for(int i = 1; i < x; i++) {
    sum += sc.nextInt();
}
System.out.println(sum);
```

Model

```java
Scanner sc =
    new Scanner(System.in);
int x = sc.nextInt();
int sum;
for(int i = 0; i < x; i++) {
    sum += sc.nextInt();
}
System.out.println(sum);
```

Snippet 3.11: Student solution does not sum the numbers, instead it subtracts.

The input of the test case needs to be valid, satisfy the pre-condition of the solution. If it does not, test cases might determine a solution to be correct when it is not and incorrect when it is correct. Thus only valid input should be tested. Automating the process of testing many different valid inputs is done by random input generation. Then feeding it to both the student solution and a model solution. The output of both solutions should be equal if the test is to pass and not equal if it fails.

Generating random input needs to be specified in a way such that it is valid. The input for the example in Snippet 3.11 firstly needs a number that is greater than or equal to zero and followed by the same number of random numbers. In the Snippet 3.12 below a random input generator for this example written in the QuickCheck DSL.

```haskell
generator :: Gen String
generator = do
  n    ← arbitrary `suchThat` (⩾0)
  nums ← replicateM n arbitrary
  return $ unwords $ map show (n:nums)
```

Snippet 3.12: A QuickCheck generator.

The first line in Snippet 3.12 is a type signature which says that the generator generates a random `String`. The next few lines define the generator. It first generates a random number that is greater than or equal to zero, then generates that many more random numbers, and finally formats all numbers as a space-separated `String`. The input needs to be a `String` as it is given to the program using `stdin` when running it through the command prompt. This is to ensure that a program has the same behaviour when automatically testing it, as it would have when manually testing it. Testing the student and model solutions using the random generator in Snippet 3.12 to generate input may yield the failing test case in Snippet 3.13 below.

```
Input: "11 18 -18 8 -4 -2 6 -5 -27 -22 10 4"
Model solution output: -32
Student solution output: -50
```

Snippet 3.13: A failing test case, as the student and model solution does not have the same output.

The failing test case in Snippet 3.13 is to a certain extent informative, it determines the student solution to be incorrect. To give the grader a better indication on what has failed the input is shrunk. The only input to a program is of the type `String` QuickCheck will always shrink it as a `String`. This means that the scheme for shrinking will be the same regardless of the generator used, and therefore may violate the pre-conditions present in the exercise specification. One approach might be to create a new type for each exercise and associate with that type a custom shrinking function which does respect the pre-conditions when shrinking. However, this approach involves a significant amount of overhead as specifying an exercise is no longer just a case of writing a simple generator. Instead an integrated shrinking approach is used with a EDSL to simplify the specification of input.

In the testing-EDSL the teaching assistant can write QuickCheck-style generators which integrate the shrinking and have primitives specific to input generation for programming exercises. The example from Snippet 3.12 can be written in the `InputGenerator` EDSL as follows:

```
generator :: InputGenerator SpaceString ()
generator = do
    n    ← anyInt `suchThat` (⩾0)
    giveInput n
    nums ← replicate n anyInt
    giveInputs nums
```

Snippet 3.14: Generate first a `Int` n, then generate n more `Int`s.

4 Implementation

This section covers how the concepts from Section 3 have been implemented. The reader is expected to have knowledge of functional programming in Haskell to fully understand the explanations in this section.

4.1 Overview

The user of the tool can configure different behaviours. The options include printing logging information at run time, specifying the input generator to be used for testing, and to only run tests when either the student solution or a model solution uses advanced Java language features not yet supported by the tool. The user also specifies the path to the student solution and the path to where the model solutions are located.

When the tool is configured it is run with the following execution steps that were previously shown in Figure 3.1:

1. Read the arguments at start and configure accordingly
2. Find the student solution in the specified path
3. Find the model solutions in the specified path
4. Use default generator if another generator has not been configured
5. Compile the student solution and model solutions
    - If it fails print the error and exit

6. Parse the student solution
    - If it fails print the error and exit, if it is specified, do not exit and continue with testing
7. Parse the model solutions, exit with error if it fails
8. Normalise all solutions
9. Match the student solution and model solutions
    - If matching fails, fallback on testing
10. Print feedback

If a solution does not compile it is by definition incorrect, since it is not a valid Java program. If it fails to parse into the AST used in the tool it has a Java feature which is not yet supported. The output is generated by the tool during execution and consists of issues and comments. Raising an issue implies that there is some problem, such as the student solution not matching any model solution. A comment indicates something positive, for instance when the student solution matches a model solution or when the student solution passes all the tests.

The tool has been implemented using Haskell. Among the advantages of using Haskell for the implementation are:

- Static typing, purity and controlled side effects - all of which simplify writing correct programs.
- Algebraic data types and pattern matching - which allows us to easily define the structures of ASTs and traverse them.
- Laziness - which in particular is instrumental for the efficient comparison of programs Section 4.3.
- Generics - which makes tree traversal and manipulation even easier as will be seen in Section 4.2.1.

4.2 Normalisation

To implement then ormalisation EDSL specified in Section 3.1.1 we have constructed the `Norm` monad. To satisfy the required properties, the monad provides the functions `unique`, `change` and `(>>=)` which are defined as follows:

- `unique, pure :: t → Norm t`, which indicates that the term t already was in normal form, and that no normalisation has occurred.
- `change :: t → Norm t`, which indicates that the term t was not in normal form, and that normalisation has occurred. If the function is used during the normalisation of any sub-branch of a term, then the term as a whole will be considered changed. This eliminates the need for explicitly writing logic that tracks change.
- `(>>=) :: Norm a → (a → Norm b) → Norm b`, which glues together smaller building blocks into a whole.

The Norm monad which uses the functions, is defined as in Snippet 4.1.

```
newtype Norm a = Norm { runNorm :: (a, Bool) }
unique, change :: a → Norm a
unique a = Norm (a, False)
change a = Norm (a, True)

instance Monad Norm where
  return  = unique
  m >>= f = let (a, u) = runNorm m
                (b, v) = runNorm (f a)
            in  Norm (b, u || v)
```

Snippet 4.1: Implementation of the normalisation monad.

To fully transform a term into unique normal form as done in Snippet 3.1, a rule, whether made of a single transformation or composed of many, is applied until it causes no change in the AST. The function `normFix` takes a rule and an AST and applies that rule on the AST until it converges at a fix-point as shown in Snippet 4.2.

```
normFix :: (t → Norm t) → t → t
normFix f t = let (t', c) = runNorm (f t) in if c then normFix f t' else t'
```

Snippet 4.2: Function for normalising a term until a fixed point is reached.

Therefore, every term must converge at a fix-point. If it does not, this has the implication that the sequence of rules [`++x ⇓ x = x + 1, x = x + 1 ⇓ ++x`] never terminates if applied to either `++x` or `x = x + 1` since the term always changes. When designing normalisation rules, caution must therefore be taken to ensure that a set of rules is reductive.

4.2.1 Bottom-up Traversals

All normalisation rules must always start from the top and traverse to the points of interest before potentially changing those regions. These points or regions are terms or sub-terms in forms which the rule wishes to transform. In the case of the Java, the entry point is a `CompilationUnit`, which represents an entire Java file in an AST [25]. The AST that is used when normalising only contains the constructs that were specified in Section 1.3. A simplified representation is given in Figure C.1.

Some terms, such as expressions (`Expr`) and statements (`Stmt`) can also be made up of sub-terms of the same type. An example of this is an `if`-statement which may contain other `if`-statements. Addition and multiplication expressions always contain two operands, which in turn are expressions. To ensure that a rule is applied on all instances, the rule is usually applied recursively on elements of the same type, which requires traversal.

To write these traversals and recursions manually is time consuming. A better strategy is to jump into any sub-term of a certain type and apply a normalisation rule to every descendant of the same type, including itself, in a bottom-up manner. To this end, the EDSL is extended with `normEvery` in Snippet 4.3.

```
normEvery :: (Data s, Data a) ⇒ (a → Norm a) → s → Norm s
normEvery = transformMOnOf biplate uniplate
```

Snippet 4.3: The function `normEvery` applies a normaliser everywhere on terms of type a within s, bottom up. The functions `transfomMOnOf`, `biplate`, and `uniplate` are described in Snippet 4.4.

The function `normEvery` applies a normalisation rule everywhere on terms of type a within a larger term of type s. The function is implemented using the traversals `uniplate`, `biplate` as shown in Snippet 4.4. These traversals are offered automatically by the lens package [26], subsuming Uniplate [27] while adding type safety.

```haskell
type Traversal' s a = forall f. Applicative f ⇒ (a → f a) → s → f s

-- | A traversal of the immediate children with the same type a.
uniplate :: Data a ⇒ Traversal' a a

-- | A traversal of all terms of type a within s.
biplate  :: forall s a. (Data s, Typeable a) ⇒ Traversal' s a

-- | Monadic bottom-up-recursive transformation with the latter
-- traversal of elements within a region specified by the former.
transformMOnOf
         :: Monad m
         ⇒ Traversal' s a → Traversal' a a → (a → m a) → s → m s
```

Snippet 4.4: Scrapping boilerplate [28] with the lens package [26].

With the necessary underlying constructs defined, normalisation rules can be implemented.

4.2.2 Using the EDSL

An example of a normalisation rule is constant folding, which is the process of evaluating constant expressions at compile time. Consider a rule which constant-folds addition and multiplication as in Snippet 4.5.

```haskell
constantFold :: Expr → Norm Expr
constantFold expr = case expr of
  EAdd (EInt l) (EInt r) → change $ EInt $ l + r
  EMul (EInt l) (EInt r) → change $ EInt $ l * r
  x                      → unique x
```

Snippet 4.5: Using the EDSL to define a normalisation rule which constant-folds addition and multiplication on the immediate level

With an example AST, visualising the use of `normEvery` applied to `constantFold` as in Snippet 4.6 of may look as in Figure 4.1.

```haskell
normEvery constantFold :: CompilationUnit → Norm CompilationUnit
```

Snippet 4.6: `normEvery` applied to `constantFold`

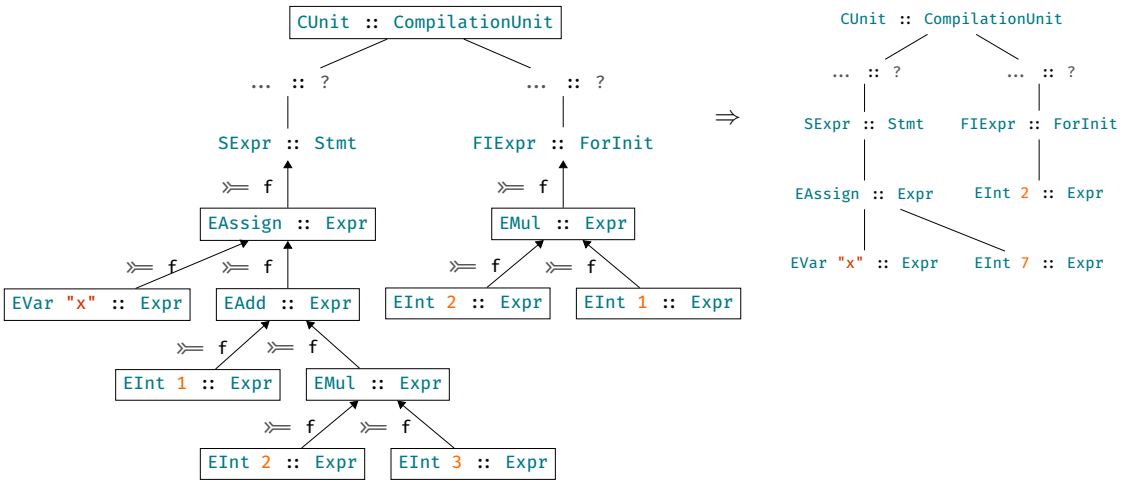

Figure 4.1: The left AST shows the AST before being normalised with `normEvery constantFold`*. All nodes with a square around them are changed when* `constantFold` *is applied. The AST to the right shows how the it looks after* `constantFold` *has been applied.*

In Java, blocks are statements containing a sequence of other statements. Therefore, blocks may be nested. Nested blocks may exist in solutions, or as a by-product of normalisation. Prior to α-renaming, blocks also introduce scoping as specified in Section 3.1.1 which prohibits concatenation of nested blocks into their parent block. However, after renaming, all nested blocks may be concatenated into a single block containing a sequence of all the statements in the original nested block.

Using the EDSL, a rule for the problem described in Section 4.2.2 called block-flattening can be implemented as shown in Snippet 4.7.

```
execFlattenBlock :: CompilationUnit → Norm CompilationUnit
execFlattenBlock = normEvery $ \b → case b of
  Block stmts → fmap (Block . concat) $ forM stmts $ \stmt → case stmt of
    SBlock (Block ss) → change ss
    s                 → unique [s]
  x           → unique x
```

Snippet 4.7: Function for executing block-flattening with the normalisation EDSL.

Locally, a `Block` goes through its immediate sequence of statements and produces a list of lists of statements. Non-block statements become singleton lists and no `change` happens, while the statements inside nested blocks are extracted in which case `change` occurs. The produced list of lists is then concatenated into a list and then boxed back into a `Block`. When `normEvery` is applied to this logic, the statements are recursively bubbled-up until only a flattened block remains. As illustrated by this example, the desired properties in Section 3.1.1 are satisfied. Additional normalisation rules that are implemented are shown in Appendix B. To implement α-renaming described of variables in Section 3.1.1, and thereby eliminate scoping, within a function definition, the following implementation scheme is used.

A map that contains the names of variables before being α-renamed mapped to their new names is referred to as a context. A stack of such contexts is kept in a `State` as seen in Snippet 4.8. The name of the next variable is kept as part of the environment in the `State` monad.

```haskell
type Context = Map Ident Ident
data Env = Env
  { stack  :: [Context]
  , nextId :: Int
  }
type Comp a = State Env a
```

Snippet 4.8: The `State` monad with `Env` defines the computional form used for α-renaming.

When a new scope is entered, a new context is pushed with `push :: Comp ()`, and when the scope is left, the context is popped with `pop :: Comp ()`. When a variable is declared, a new substitution is added to the top context in the stack with `newMapping :: Ident → Comp Ident`, and the variable is renamed at the declaration site. When a variable is used, the stack is searched top to bottom for the first mapping where the name of the variable matches $oldName$, and is then substituted for $newName$. This substitution is retrieved with `substitute :: Ident → Comp Ident`.

A specification for a student exercise might not explicitly define what classes and functions should be named. Therefore, to allow students to use arbitrary names for classes and functions, they are also renamed using a method similar to variable renaming. However, imported classes and functions are not renamed since they are unknown to the tool.

4.2.3 Naming Normalisers

To increase usablility of the tool and to make it easier for a teacher to understand what a normalisation rule does, the actual functions that use the EDSL and encode the logic for rules are named as shown in Snippet 4.9.

```haskell
-- | A named normalisation rule.
data NamedNRule a = NamedNRule
  { name    :: String -- ^ A machine readable key for the rule.
  , execute :: Norm a -- ^ The logic for the rule.
  }

normFlattenBlock = NamedNRule "elim_redundant.stmt.flatten_block"
                              execFlattenBlock
```

Snippet 4.9: Naming the rule in Snippet 4.7.

The `name` can then be used to represent the normalisation. This key can then be translated into longer descriptions and formats more presentable to a teacher. Rules are also grouped together in a hierarchical manner where each each level is separated with a dot (.). In the example shown in Snippet 4.9, the first hierarchy describes a large class of normalisers, the second describes that it is a statement, while the third is unique specific rule.

4.3 Matching

This section describes the process of matching a student solution against a model solution. As outlined in Section 3.2.2 the process happens in two stages. First a prefix tree is constructed from the model solution, then the student solution is matched against that prefix tree.

4.3.1 Generating Prefix Trees

Prefix trees are represented as a tree of Java ASTs, seen in Snippet 4.10. To simplify the construction of prefix trees a customised `AST` representation is used. The feature that differs in this `AST` and the other internal representation is that it is unityped, meaning that all constructors have the same

type. The type contains the constructs of the Java language supported by our tool, extended with a constructor for holes. The presence of a hole is what makes an AST a prefix.

```
data PrefixTree = Node AST [PrefixTree]
```

Snippet 4.10: Definition of the data structure PrefixTree.

The generation of the prefix tree is based on the principle of replacing a hole with another prefix `AST`. This means that the process needs to start by replacing each subtree with holes, while simultaneously describing how to put that `AST` back in the place of the hole. The instruction of how to put the `AST` in the place of a hole is called hole-refinement. The process of replacing `AST`s with holes is done in a bottom up manner, which means that each subtree will have as many parts of itself replaced with holes as possible before being replaced itself.

The instructions for how a `PrefixTree` is created is defined as a `Strategy AST` using the `Ideas.Common.Strategy` module, a part of the IDEAS framework [29]. The smallest building block of the `Strategy AST` defines the execution for one concrete step in the process of building an `AST` and can be combined using combinators. In the tool these steps are the hole-refinements. While more combinators exists, the tool only uses the succession combinator, `(.*.)`, and the choice combinator, `(.|.)`.

Besides being able to reconstruct a given `AST` we need the instructions for how to create all semantically equivalent permutations of that `AST`. We call these instructions *dependency-strategies*. The creation of the dependency-strategy is done in tree steps. Given the `AST`s to be ordered and a function `dependsOn :: AST → AST → Bool`, which implements the dependency analasis described in Section 3.2, we start by creating a dependency-DAG. This DAG has nodes representing each `AST`, and have edges from each node to each node representing an earlier `AST` on which it depends. Using this DAG we construct a tree of `AST` representing each possible topological ordering of the `AST`s. The levels in the tree represents one position in the order, and each pathway represents one possible order. Finally, this tree is converted into a `Strategy AST` using the combinators. This process is visualised in Figure 4.2.

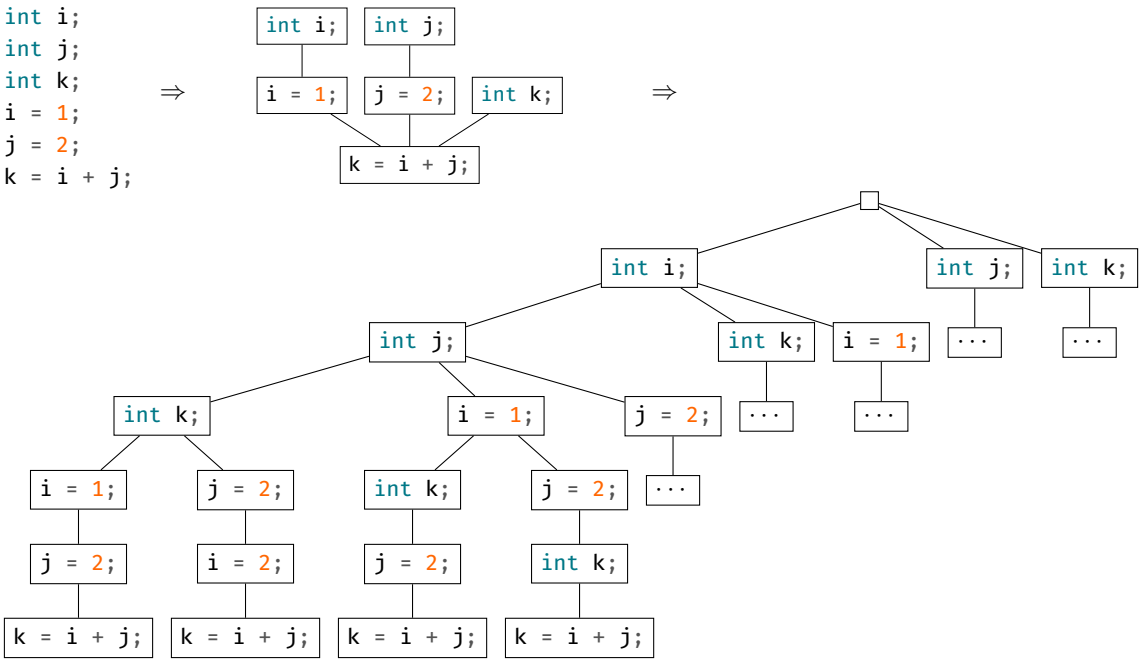

Figure 4.2: The steps required to transform a solution to a tree representing all topological orderings.

Once the `Strategy AST` is created it can be used to generate the `PrefixTree`.

4.3.2 Matching Using Prefix Trees

Once a prefix tree has been obtained a student solution may be compared to it to establish if there exists a permutation of the model solution which is equivalent to the student solution. As described in Section 3.2 the partial programs in the prefix tree need to be alpha renamed, this is done using the function `rename`. The procedure for matching is given as the `matches` function, given in Snippet 4.11.

```
matches :: PrefixTree → AST → Bool
matches tree ast = go [tree]
    where
        go []                              = False
        go ((Node a []):trees)
            | ast == (rename a)            = True
            | otherwise                    = go trees
        go ((Node _ [a]):trees)            = go (a:trees)
        go ((Node a branches):trees)
            | (rename a) `isPrefixOf` ast = go (branches ++ trees)
            | otherwise                    = go trees
```

Snippet 4.11: A function for matching a student solution against a prefix tree generated from a model solution. The function `rename` applies the α-renaming normalisation rule to an `AST`.

Note the final clause in the `go` function, it discards all children of a prefix node which is not a prefix of the AST we are trying to match. Other than that the function is a standard depth first search.

4.4 Testing

This section describes the process of testing a student solution against a model solution. As outlined in Section 3.3 the teacher creates a generator using the testing-EDSL. This generator is used to feed input, test cases, to the solutions. If a test fails, the input is shrunk to find a 1-minimal test case.

The method for testing requires compiling the solutions with the `javac` compiler and running it using the `java` program. This ensures that the solution behaves the same way during testing as it does if a grader were to run it manually. It also means that all input is given to the program at start or using `stdin`, and the output must be printed to `stdout`. This implies that all input and output must be of the type `String`.

4.4.1 Generation of Input

The implementation of the EDSL for writing input generators described in Section 3.3 is shown in Snippet 4.12.

```
import qualified Test.QuickCheck as QC

-- | Generator wrapping the Generator from QC
newtype Generator a = Generator { unGen :: QC.Gen (Tree a)}
```

Snippet 4.12: Implementation of `Generator`.

Several functions to generate input are provided to give familiarity with QuickCheck and make the implementation as intuitive as possible. The three main functions are those show in Snippet 4.13.

```
-- | Generate an arbitrary value, and all ways to shrink that value
arbitrary :: (QC.Arbitrary a) ⇒ Generator a

-- | Generate a value such that a predicate holds for that value
-- | and the predicate holds when shrinking
suchThat :: Generator a → (a → Bool) → Generator a

-- | Run the generator, generating a Tree
generate :: Generator a → IO (Tree a)
```

Snippet 4.13: Main functions used for generating input.

With these functions the user can construct a generator for arbitrary values, such that a predicate holds. The `suchThat` function is used to specify the predicate for generating and shrinking. When the tool runs `generate`, the function returns a tree where the root is the value generated and the children are how it can be shrunk, both satisfying the predicate.

4.4.2 The Testing EDSL

The EDSL is implemented as the monad `InputGenerator m` shown in Snippet 4.14 which uses `WriterT` that takes a monoid, specifically the monoid `InputMonoid`, to specify how the functionality of the monad works.

```
type InputGenerator m a = WriterT m Generator a
```

Snippet 4.14: The monad implementing the EDSL.

The `InputMonoid` constraints specifies the functionality of the `InputGenerator`, by using a `Wrapper` that implements two functions called `wrap` and `unwrap`, as shown in Snippet 4.15[1].

```
type InputMonoid m = (Wrapper m String, Monoid m)

class Wrapper m a where
  wrap   :: a → m
  unwrap :: m → a
```

Snippet 4.15: The special input monoid and the class.

An example of an `InputMonoid` is the type `NewlineString` in Snippet 4.16. The difference between a `NewlineString` and `String` is that the `Monoid` instance for ordinary `String`s use concatenation, `(++)`, as their `mappend` operation, while `NewlineString`s insert a newline character `'\n'` between the operands of `mappend`.

[1]The code in this figure relies on the *ConstraintKinds* language extension

```haskell
newtype NewlineString = NL { unNL :: String }

instance Monoid NewlineString where
    mempty = NL ""

    (NL "") `mappend` x = x
    x `mappend` (NL "") = x
    x `mappend` y       = NL $ unNL x ++ "\n" ++ unNL y

instance Wrapper NewlineString String where
    wrap   = NL
    unwrap = unNL
```

Snippet 4.16: The `NewlineString` monoid

The functions `giveInput` and `giveInputs`, as shown in Snippet 4.17, wraps the generated value. The generated value needs to be able to cast to `String`. To make an abstraction the functions casts the value and wraps it.

```haskell
-- | Take a value a, cast it to a String, then wrap it
giveInput :: (InputMonoid m, Show a) ⇒ a → InputMonad m ()
giveInput a = tell $ wrap $ show a

-- | wrap a list of values
giveInputs :: (InputMonoid m, Show a) ⇒ [a] → InputMonad m ()
giveInputs list = mapM_ giveInput list
```

Snippet 4.17: The `giveInput` and `giveInputs` functions.

Using the `InputGenerator` with the example: read a number n, read n numbers, is shown in Snippet 4.18 below.

```haskell
exercise0 :: InputMonoid m ⇒ InputMonad m ()
exercise0 = do
  n ← (arbitrary :: Generator Int) `suchThat` (\x → x ⩾ 0)
  giveInput n
  numbers ← replicateM n (arbitrary :: Generator Int)
  giveInputs numbers
```

Snippet 4.18: Generate a number n, then generate n numbers.

To extract the `Generator` from the `InputMonad` the tool calls the function `makeGenerator` as shown in Snippet 4.19. The function runs the `WriterT` and unwraps the `InputMonoid` to make the `Generator String` which is used to generate input to the program under test. Note that the use of the `InputMonoid` constraint provides sufficient generality that the same exercise specification may be used with different separators between input given by `giveInput`, using `NewlineString` to separate items by newline characters, or a type like `SemicolonString` to separate by semicolons.

```haskell
-- | Construct a `Gen String` from an `InputMonad a`
makeGenerator :: InputMonad m a → Generator String
makeGenerator input = unwrap <$> runWriterT input
```

Snippet 4.19: Run the Writer with the input and make a `Generator String`.

4.4.3 Shrinking

The root of the generated tree is the input to the program that is being tested. Once a failing root input is found a 1-minimal failing child of that root test case should be found in the tree to shrink

the input. Every child of the root is a tree with the same structure. Searching for a 1-minimal failing test case in the tree is done by a depth first search algorithm. It searches the left most child, until it reaches the 1-minimal failing test case. The example in Figure 4.3 shows a tree of only even numbers, where the original test has failed for the root input of 8. The children are all the possible ways to shrink that input. As Haskell is lazy the whole tree will not be generated, but only the children that the tool tests. If no leaf node is found, the failing test case deepest in the tree is chosen as the 1-minimal failing test case.

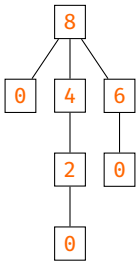

Figure 4.3: A tree with the root 8 and all the ways to shrink it.

The children in the tree are ordered in ascending order which helps finding the 1-minimal failing test case fast as the searching algorithm ensures that the left most child is tested first.

5 The `Javista` Tool

This section gives an overview of the usage of `Javista` command line interface. The user of the tool specifies the path to the student solution, the path to the folder where all the model solutions are located, and possible optional commands. The optional `-h` command gives a brief overview of how to use the tool, the output generated when running with `-h` can be seen in Figure 5.1.

```
$> Javista -h
Javista, a program for Java Automated Assessment
Usage: Javista STUDENT_SOLUTION MODEL_SOLUTIONS_DIR [-g|--generator TEST_GENERATOR]
               [-v|--verbose] [-l|--logfile LOGFILE] [-n|--numTests NUM_TESTS]
               [-i|--ignoreFailingParse]
Available options:
  -g,--generator TEST_GENERATOR
                           Should be on the form module:generator
  -v,--verbose             Prints log messages during execution
  -l,--logfile LOGFILE     Logfile produced on program crash
  -n,--numTests NUM_TESTS  Number of tests during property based testing
  -i,--ignoreFailingParse  Ignore the Javista parser failing if javac was OK,
                           proceed with testing only
  -h,--help                Show this help text
```

Figure 5.1: Output printed when running `Javista` with the `-h` flag

The flags that can be set at start to customise the behaviour of the tool. Figures 5.2, 5.3, and 5.4 below shows some examples of the tool running.

```
$> Javista -v stud.java mods
Checking if stud.java exists
Checking for model solutions in directory "mods"
Found the following model solutions in directory "mods": mod1.java
Using "arbitrary" generator
Running the command: javac -d compilationDirectory/model mods/mod1.java
Running the command: javac -d compilationDirectory/student stud.java
Reading student solution
Reading model solutions
Parsing student solution
Parsing model solution: mods/mod1.java
Matching student solution to model solutions
Checking: mods/mod1.java
Comments:
0. Student solution matches a model solution: mods/mod1.java
Issues:
No issues :)
```

Figure 5.2: Output printed when running `Javista` *with the* `-v` *flag*

Using the `-v` option, as shown in the example in Figure 5.2, makes the `Javista` print logging information during execution. The final output in Figure 5.2 clearly states that the student solution matched a model solution, the solution is assessed as correct.

```
$> Javista stud.java mods
Comments:
No comment...
Issues:
0. Student solution does not match a model solution
1. Failed on input: Input [] ""
    With
    Student solution output: -0.0039999991984016257
    Model solution output: 3.19592655589785
```

Figure 5.3: A student solution where `Javista` *found a fails testing*

When a student solution does not match any of the model solutions, the tool fallback to testing. The tool then tests the student solution against the model solutions the number of times that are specified at start, or the default 100 times. If a failing test case is found, the 1-minimal test case is printed as an issue, as is shown in Figure 5.3. This solution is assessed as incorrect.

```
$> Javista stud.java mods
Comments:
0. Student solution passed all tests
Issues:
0. Student solution does not match a model solution
```

Figure 5.4: A student solution which does not match a model solution and passes all tests

A student solution that passes all tests and does not match a model solution is assessed as unclassified, also shown in Figure 5.4.

6 Results

The work presented in this report has resulted in a number of contributions, including the EDSLs for writing normalisation rules and input generators. However, the primary contribution is the prototype `Javista` tool and its underlying architecture described in sections 3 and 4. As a proof of concept we have used `Javista` to automatically assess 47 real student solutions to an exercise from the course `TDA540` at Chalmers university of technology. The assignment was to calculate π using the Leibniz formula with the first 500 terms and display the result. The full exercise specification can be found in Appendix A. Because the tool does not support GUI-programming the solutions using graphical components to display the result of the computation were manually rewritten to to print the result to `STDOUT` instead. Furthermore, some solutions displayed more than the result of the computation, e.g. printing `"Pi is: 3.14 ..."` instead of just `"3.14 ..."`, these programs were manually rewritten to only print the final result. Manual assessment of the 47 student solutions showed that 22 were incorrect and 25 were correct. The results of running `Javista` on this dataset are summarised in Table 1. `Javista` assessed all incorrect solutions as incorrect, 5 of the 25 correct solutions as correct, and the other 20 correct solutions as unclassifiable.

		Manual	
		Correct	Incorrect
`Javista`	Correct	5	0
	Incorrect	0	22
	Unclassifiable	20	0

Table 1: Results of the experiment

Manual comparison of correct student solutions wihch were assessed as unclassifiable to model solution revealed multiple examples of student solutions which were only subtly different from one or more of the model solutions. This is an indication that adding further rewrite rules to the normalisation stage would be helpful in increasing the total number of matched student solutions. One example of such a subtle difference is found in Snippet 6.1.

Model

```
for (int i = 0; i < 500; i++) {
  ...
}
```

Student

```
for (int i = 1; i ⩽ 500 ; i++) {
  ...
}
```

Snippet 6.1: A student solution which was not matched by the tool

7 Discussion

From the results presented in Section 6, it can be determined that the tool is a step towards automated assessment. In fact, 60% of the student solutions in the evaluation were correctly assessed with no false positives. Thus our result shows that *Javista* achieved our goal of aiding a teacher in their task of assessing student solutions. Furthermore, the fact that the tool did not yield any false positives, as a result of all normalisation rules being semantic preserving, implies that the goal of building a reliable tool was also achieved.

Keuning et. al. [24] presents results of 33% - 75% recognition of solutions. In their case, the solutions were integrated into the tool which supplied guidance during the solving process and the assignment was specified in a way that leaves less room for variations.

Making assignments that are more suited for the capabilities of the tool would increase the number of recognised solutions. By using an existing data-set we could improve the tool by analysing why it does not recognise a specific solution and implement relevant features to recognise it. Using this method the tool would hopefully also improve in a more general case.

As indicated by the example in Snippet 6.1, further transformation rules must be added to the normalisation procedure to match more solutions. The modularity of the tool as well as the usability of the normalisation EDSL, makes creating new normalisation rules relatively simple.

7.1 Normalisation

Since solutions can use varying syntax, some transformations that retain semantic equivalence are essential for static analysis. We chose to use normalisation for this since it reduces the amount of variations that can occur in a solution.

The normalisation EDSL was implemented with the goal of simplifying the creation of transformation rules. The EDSL increases modularity by making it easy to create further normalisation rules and combining them. There are alternative ways to implement the normalisation EDSL described in Section 3.1.1. Two candidates for the normalisation monad are:

- The `Identity` monad with normaliser functions of type `Eq a ⇒ a → a`. This monad requires tracking change manually by testing for equality, which negatively affects performance.
- The `Maybe` monad with normaliser functions of type `a → Maybe a`. In the context of normalisation, if applying a normaliser on a term yields `Nothing`, then it was already in normal form. `Maybe` does not encode the desired semantics, instead, it encodes the opposite: *there was a change at the top if there was a change in all sub-terms.*

While normalisation reduces the number of possible variations, normalisation could result in accepting student solutions that should not be accepted. The student may have written a solution that while correct, is far too complicated but is matched to a model solution after normalisation. This would mean that the teacher would still have to look at the solution to ensure that it is not written in a bad way.

Currently, all normalisation rules are always enabled, meaning that there is no way disable unwanted ones. Removing unwanted rules could be meaningful for a teacher if the exercise given is supposed to teach the students about `for` loops. The tool transforms all `for` loops into `while` loops which means that a student solution using `while` loops would be considered correct by the tool.

7.2 Matching

We introduced the matching as a way to define equality between programs that could not be transformed to the same unique normal form. We chose to implement the matching as using the `Ideas`-package and prefix trees. In order to support this a rather complex structure had to be made. Due to the fact that the ordering problem was the only problem that was solved using the matching, it is uncertain if the chosen approach was efficient for its use. It is however plausible that the structure simplifies the addition of further solutions to problems that would be difficult to solve with normalisation.

7.3 Testing

A large number of the student solutions used in the evaluation of the tool provided more output than required by the exercise specification. While the specification said to print an approximation of π, many student solutions either used a simple GUI to provide the output or printed more text than specified, e.g. `"Pi is 3.14 ... "` rather than just `"3.14 ... "`. Solving this problem could have been done by analysing the student solution and making a unified way of showing the result. But this was not in the scope of the project and the issue can be solved in other ways, such as that the assignment specifies how the result should be shown, in our case it should be printed to `stdout` in a certain way. There are some possible alternatives which we could have implemented given the time. One alternative is to augment the testing-EDSL with a way of specifying what it means for solution's output to be correct. That way a generator could for example specify an interval of valid

outputs for π in our example above, or search for a valid value of π in the output. This method could effectively get rid of the model solutions for testing, but may make writing generators more difficult.

Another method to run the tests that where considered was to implement an interpreter. However, this would not be superior if the scope expands, since it would require a very complex interpreter.

To improve testing, more properties could be added to the tool. At the moment, the only property that is tested against are the outputs of model solutions. Specifying properties that hold could be done much in the same way as specifying the generator. Allowing testing of ore properties would improve the quality and quantity of information that the tool could provide a teacher.

7.4 Javista in Society

The tool can be used to aid teachers and teaching assistants in a course to shift the time and money allocated to grading and correcting exercises. Instead the teachers and teaching assistants could aid students in the process of problem solving. This could improve the working environment for students as they would get more support and help.

Automated assessment is particularly well suited for distance learning. An example would be to use the `Javista` tool in Massive Open Online Courses (MOOC), where anyone can enroll. By eliminating the time it takes for the teacher to grade solutions, more students could be enrolled into these courses. If successful, the tool could possibly reduce the overall cost of education.

There are no immediate global environmental effects that the *Javista* tool has, that we can see.

8 Conclusion and Future Work

The purpose of this project was to develop a tool to aid in the assessment of student programming exercises by automatically assessing them. To this end, the tool used normalisation, matching, and testing. Matching was introduced to cover the parts that were unfeasible to cover with normalisation while the testing part of the tool was stand-alone and was developed separately.

Automatically assessing student solutions is a difficult problem. The goal was to make a useful and reliable tool that could assess a student solution as either, correct, incorrect, or unclassifiable. The results show that the `Javista` tool is both useful and reliable. With 60% of the student solutions assessed as either correct or incorrect with no false positives. Furthermore, 100% of the incorrect solutions where assessed as incorrect by the tool, while 20% of the correct solutions where assessed as correct. This indicates that there is room for improvement in the static analysis part of the tool.

Analysing what type of statements and expressions that the matching has difficulty with and directing the creation of new normalisation rules towards the failing cases would increase recognition. In the case of exercise 12 in the appendix, adding constant folding would be particularly useful. As the tool has been developed with modularity in mind, it can be easily extended with more normalisation rules, more descriptive output, and a more thorough testing suit.

In conclusion we believe that it is possible to make a tool for automated assessment of Java. It will, however, require some future work, including the items in the list below:

- The implementation of `Javista` currently only supports basic Java constructs. Extending the tool to support the entire Java language requires significant engineering effort, but is possible.
- Since Java is a very open ended language, there is almost no limit to the number of normalisation rules that can be added. Due to the modular implementation of the normalisation EDSL this is likely to be a straight forward task.

- The testing EDSL could be enriched with specification of different generators for the same exercises and what methods those generators generate input for. This could be used in combination with properties for each generator, thus giving the grader a better understanding of what fails.
- Instead of matching one solution at a time the tool could use a whole directory of student solutions and run them in parallel. Also if a whole directory is used, the testing could group solutions that fails with the same property, with inspiration from the paper about ranking solutions using black box testing [30].
- The interaction with the tool could be extended with more flags and exhaustive output
- In order to indicate that a solution is overly complex compared to a model solution, complexity analysis [1] and cyclomatic complexity analysis [31] could be used.
- Currently the tool does not do any type inference on student or model solutions. Including type inference would enable implementing certain normalisation rules such as transforming floating point literals, such as `1.0` to integer literals `1` where appropriate.
- Annotated model solutions in the style of [14] which allow teachers to customise the matching and output generation processes could be implemented as a step towards providing feedback directly to students.

References

[1] C. Benac Earle, L.-Å. Fredlund, and J. Hughes, ''Automatic Grading of Programming Exercises using Property-Based Testing,'' in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ACM, 2016, pp. 47–52. DOI: 10.1145/2899415.2899443.

[2] D. Insa and J. Silva, ''Semi-automatic assessment of unrestrained Java code: a library, a DSL, and a workbench to assess exams and exercises,'' in *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ACM, 2015, pp. 39–44. DOI: 10.1145/2729094.2742615.

[3] B. Heeren, J. Jeuring, and A. Gerdes, ''Strategies for Exercises,'' Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2009-003, 2009. DOI: 10.1007/s11786-010-0027-4.

[4] H. Keuning, B. Heeren, and J. Jeuring, ''Strategy-based feedback in a programming tutor,'' in *Proceedings of the Computer Science Education Research Conference*, ACM, 2014, pp. 43–54. DOI: 10.1145/2691352.2691356.

[5] J. Jeuring, A. Gerdes, and B. Heeren, ''Ask-elle: A haskell tutor,'' in *European Conference on Technology Enhanced Learning*, Springer, 2012, pp. 453–458. DOI: 10.1007/978-3-642-33263-0_42.

[6] K. Arnold, J. Gosling, D. Holmes, and D. Holmes, *The java programming language*. Adtdison-wesley Reading, 2000, vol. 2.

[7] A. Gerdes, *Tda540 - objektorienterad programmering*, 2015. [Online]. Available: http://www.cse.chalmers.se/edu/year/2015/course/TDA540/Lectures/ (visited on 02/02/2017).

[8] P. Hudak, ''Building domain-specific embedded languages,'' *ACM Computing Surveys (CSUR)*, vol. 28, no. 4es, p. 196, 1996. DOI: 10.1145/242224.242477.

[9] B. C. Pierce, ''Types and programming languages,'' in. MIT Press, 2002, ch. Section 5.1 Basics, Abstract and Concrete Syntax, pp. 53–54, ISBN: 0-262-16209-1.

[10] D. Sannella and A. Tarlecki, ''On observational equivalence and algebraic specification,'' *Journal of Computer and System Sciences*, vol. 34, no. 2, pp. 150–178, 1987, ISSN: 0022-0000. DOI: http://dx.doi.org/10.1016/0022-0000(87)90023-7. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0022000087900237.

[11] S.-y. Katsumata, ''Behavioural equivalence and indistinguishability in higher-order typed languages,'' in *Recent Trends in Algebraic Development Techniques: 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, September 24-27, 2002, Revised Selected Papers*, M. Wirsing, D. Pattinson, and R. Hennicker, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 284–298, ISBN: 978-3-540-40020-2. DOI: 10.1007/978-3-540-40020-2_16. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-40020-2_16.

[12] W. Landi, ''Undecidability of static analysis,'' *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, Dec. 1992, ISSN: 1057-4514. DOI: 10.1145/161494.161501. [Online]. Available: http://doi.acm.org/10.1145/161494.161501.

[13] B. C. Pierce, ''Types and programming languages,'' in. MIT Press, 2002, ch. Section 21.1 Metatheory of Recursive Types, Induction and Coinduction, p. 282, ISBN: 0-262-16209-1.

[14] A. Gerdes, J. Jeuring, and B. Heeres, *Using strategies for automated assessment of programming exercises*, 2009. DOI: 10.1145/1734263.1734412.

[15] M. Huth and M. Ryan, ''Logic in computer science: Modelling and reasoning about systems,'' in, "second". New York, NY, USA: Cambridge University Press, 2004, ch. Section 1.2, Natural deduction, p. 27, ISBN: 9780521543101.

[16] E. W. Dijkstra, ''Notes on structured programming,'' in. Technological University Eindhoven, Deptartment of Mathematics, 1970, p. 5.

[17] ——, ''Notes on structured programming,'' in. Technological University Eindhoven, Deptartment of Mathematics, 1970, p. 7.

[18] K. Claessen and J. Hughes, ''Quickcheck: A lightweight tool for random testing of haskell programs,'' *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011. DOI: 10.1145/1988042.1988046.

[19] A. Zeller and R. Hildebrandt, ''Simplifying and isolating failure-inducing input,'' *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[20] Ambiata, *Disorder-jack*, 2016. [Online]. Available: https://github.com/ambiata/disorder.hs/tree/master/disorder-jack (visited on 05/25/2017).

[21] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java® Language Specification*, 2015. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/index.html (visited on 05/09/2017).

[22] R. Wilhelm, H. Seidl, and S. Hack, "Compiler design: Analysis and transformation," in, 1st. Springer Publishing Company, Incorporated, 2012, ch. Section 1.1, Introduction, p. 4, ISBN: 9783642175473. DOI: 10.1007/978-3-642-17548-0.

[23] B. C. Pierce, "Types and programming languages," in. MIT Press, 2002, ch. Section 5.3 The Untyped Lambda-Calculus, Formalities, p. 71, ISBN: 0-262-16209-1.

[24] H. Keuning, "Strategy-based feedback for imperative programming exercises," Master's thesis, Open Universiteit Nederland, 2014, p. 43.

[25] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java® Language Specification, Chapter 18. Syntax*, 2015. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/jls-18.html (visited on 05/31/2017).

[26] E. A. Kmett, *Lens: Lenses, folds and traversals*, 2012. [Online]. Available: https://hackage.haskell.org/package/lens (visited on 05/06/2017).

[27] N. Mitchell, *Uniplate: Help writing simple, concise and fast generic operations*, 2006. [Online]. Available: https://hackage.haskell.org/package/uniplate (visited on 05/06/2017).

[28] R. Lämmel and S. P. Jones, "Scrap your boilerplate," ACM, 2003, pp. 26–37. DOI: 10.1145/604178.604179.

[29] B. Heeren, A. Gerdes, and J. Jeuring, *Ideas: Feedback services for intelligent tutoring systems*, 2009. [Online]. Available: https://hackage.haskell.org/package/ideas (visited on 05/08/2017).

[30] K. Claessen, J. Hughes, M. Pałka, N. Smallbone, and H. Svensson, "Ranking programs using black box testing," in *Proceedings of the 5th Workshop on Automation of Software Test*, ACM, 2010, pp. 103–110. DOI: 10.1145/1808266.1808282.

[31] A. Sobey, *1.3 basis path testing*, 1995. [Online]. Available: http://users.csc.calpoly.edu/~jdalbey/206/Lectures/BasisPathTutorial/index.html (visited on 02/02/2017).

A Exercise 12 - Translated from Swedish

The famous mathematician Gottfried Leibniz gave the following formula for π:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots$$

- Write a program that computes π according to this formula and includes the first 500 terms. The method `Math.pow` is not allowed in the solution.

  Tip The expression is a sum. A sum is a series of terms that are added together. In our current expression the terms consist of a numerator and a denominator. A term can be computed from the previous term. The denominator is 2 larger than in the previous term. The absolute value of the numerator is always 1, but the sign is flipped in every term (i.e: alternating between positive and negative).

- Write a program which compute π according to this formula and includes as many terms that the absolute value of the last included term is smaller than 0.00001.

B Implemented Normalisation Rules

- α-renaming.
- If index of `for` loop begins at 1 and is not used in the loop, transform it to 0 and change comparatorr from $\leqslant$ or $\geqslant$ to < or >.
- Transform a `for` loop to `while` loop.
- Transform compund assignment into regular assignment with expression on right hand.
- Move variable declarations and initiations out from `for` loop.
- Split variable declaration with many variables into many variable declarations.
- Move array dimension in declaration from right of variable to left of variable.
- Split variable declaration and initiation. `int x = 1;` => `int x; x = 1;`
- Move all variable declarations to the top.
- Flatten statement blocks into their parent blocks.
- Remove empty blocks.
- Filter empty statements.
- If a block contains a single statement, remove the block around it.
- Remove dead `if`-statements.
- Remove dead `do`-statements.
- Remove dead `while`-statements.
- Remove dead `for`-statements.
- Transform `do while` loop into `while` loop.
- Remove empty `if` in `if-else` statement.
- Remove empty `else` in `if-else` statement.
- Remove empty `if` and `else` in `if-else` statement.
- Transform `i++` in `for` loops into `i = i + 1`.
- Transform `i++` in statements into `i = i + 1`.
- Transform `i++` in expressions into `i = i + 1`.
- Transform `float` variables into `double` variables.
- Transform methods with return type `float` into `double`.
- Transform numerical expressions into sum of products form.
- Transform `for(int i = x; i ⩽ y; i++)` to `for(int i = x; i < y + 1; i++)`

C AST Definition

```
data CUnit = CompilationUnit [TypeDecl] -- ^ Entry point

data TypeDecl = ClassTypeDecl ClassDecl -- ^ Class type declaration

data ClassDecl = ClassDecl Ident ClassBody -- ^ Class declaration

data ClassBody = ClassBody [Decl] -- ^ Class body

data Decl = Decl MemberDecl -- ^ Class members

data MemberDecl = MethodDecl Type Ident [FormalParam] Block -- ^ Method declaration

data FormalParam = Type Ident -- ^ Method parameters

data RType = Void
           | RType Type

data Type = BoolT -- ^ Types
          | IntT
          | StringT
          ...

data Block = Block [Stmt] -- ^ Blocks

data Stmt = SBlock Block  -- ^ Statements
          | SExpr Expr
          | SIf Expr Stmt
          | SWhile Expr Stmt
          | SFor FIExpr Expr [Expr] Stmt
          | SDecl Type
          ...

data Expr = EInt Integer  -- ^ Expressions
          | EVar Ident
          | EAssign Ident Expr
          | EAdd Expr Expr
          | EMul Expr Expr
          ...
```

Figure C.1: Part of the implementation of the AST.