# HOLMES: Hunting for Haskell Frauds

B.M. Vermeer, 3122212
bmvermee@students.cs.uu.nl

Master of Science Thesis

Supervisor: Dr. J. Hage
jur@cs.uu.nl

Center for Software Technology,
Department of Information and Computing Sciences,
Utrecht University,
P.O.Box 80.089, 3508 TB,
Utrecht, The Netherlands

February 2010

*To ...*

# Contents

# Acknowledgements

I would like to thank everyone who made a contribution to my Master study "Software Technology" at the Department of Information and Computing Sciences of Utrecht University. The professors, lecturers, student and friends made my time at Utrecht University unforgettable.

A special thanks to my supervisor Jurriaan Hage, for assisting me during this project. All the brainstorming, comments and suggestions helped made this Master Thesis project what it is now. I want to thank him for all the time he donated to me and this project. In the meetings we had during this project I always admired his enthusiasm for this project.

My time during my Master study made me look, think and reason in whole new way if I compare this to my former study. I want to thank all the lecturers I had during my time at Utrecht University for that.

I want to end this acknowledgement by thanking my family and close friends for believing and supporting me during my study and in particular this thesis project.

# Abstract

For the functional programming language Haskell there is no specific tool available to compare source code for plagiarism. Other, more used, programming languages like Java do have tool support for checking plagiarism. Especially for educational institutes it would be convenient to have tool support for checking large batches of submissions for plagiarism. When checking Haskell submission for plagiarism it is important first to discover how we can achieve that. What do we need to compare, how do we compare it and can we automate this procedure. A possible solution to compare Haskell programs for plagiarism was already known at the beginning of this thesis. There is a tool called MOSS that can compare Haskell programs. However it is based on a universal technique rather then specifically designed for Haskell and therefore doesn't use the specific characteristics of Haskell. This thesis focuses on the issue how to detect possible plagiarism in Haskell submission by using the characteristics of the language. Therefore we created a tool, based on Helium, that parses the source and applies various heuristics to compare the sources. This program, called Holmes, consists of a pre-process that normalises the source and a compare tool that compares the normalised sources. The implemented heuristics divided in three categories: structural, semantic and literal. All heuristics are applied to both prepared and unprepared test sets to both verify and validate the outcome. At the end of this project the outcome of the heuristics implemented in Holmes showed us that only a few heuristics give useful results looking for plagiarism in Haskell sources. The pre-processor turns out to be a very important link in this process. The implementation shows that we can automatically detect possible software plagiarism in a functional languages like Haskell.

# Chapter 1

# Introduction

The popularity of functional programming grows more and more. Prominent functional languages, and Haskell in particular, have not only been used within the context of science but also in industrial and commercial applications. With the growing popularity of languages like Haskell comes that programmers that used to work in other languages may switch. Universities and other educational institutes will be, or are already, teaching functional languages.

This growing popularity comes with a realistic problem. When more people use functional languages like Haskell, occurrences of fraud by plagiarism will be present and will be harder to detect. An example is when a programmer, for example a student, takes other peoples programs and claims to be the author.

## 1.1 Problem Description

Plagiarism is a serious issue, not only for commercial programs, but also and perhaps even more, within the education system. In the computer science department at Utrecht University plagiarism and other forms of fraud are taken very seriously. Every year there are a handful of cases in courses where Java is the main language that are considered to be plagiarism. We have no reason to believe that this will be different for courses where Haskell is the main language. For plagiarism detection in Java source code several tools are available. At Utrecht a tool called Marble has been developed to cross compare Java submissions. In practise it turns out that Marble works quite well for Java, but a small experiment to compare Haskell submissions with it failed however.

Within the educational system an assignment for Haskell may be handed out to large number of students. An assessor usually wants to prevent that plagiarised submissions will be considered valid. Manual inspection for plagiarism on large amounts of similar submissions is simply infeasible particularly if we have to

consider submissions over multiple years. For an assessor it would therefore be convenient to automatically check Haskell submissions for plagiarism.

## 1.2 Contribution

There are many forms of plagiarising a submission. A student can completely copy a submission resulting in an identical program or, as happens in many cases, a student can try to hide plagiarism by changing the original version. In this thesis we discuss how we tried to detect possible plagiarism in Haskell. We defined a diverse set of heuristics to inspect Haskell submissions similarly on various aspects and provide algorithms to compare two Haskell submissions. We also describe how we should handle a large amount of submissions.

To test the described heuristics we developed a tool called Holmes. Holmes consists of a pre-processing part and a comparison part. Holmes is based on the Helium compiler and therefore covers a large subset of Haskell 98. We implemented all heuristics in this tool and validated the scores on actual submission from the FP (Functional Programming)[1] course.

The intention of this project was not to build a tool that can be widely used. The tool we built was designed as an instrument to discover if we can detect plagiarism in Haskell by automatic detect and which heuristics contribute to doing so. The Holmes system, although useable for Helium compatible programs, is only a side effect of this project.

## 1.3 Outline

The thesis is structured as follows. We first discuss the research context the and notions on plagiarism and fraud in chapter 2. We then explain, in chapter 3, the approach including the requirements we determined for Holmes. The high level architecture of Holmes is described in chapter 4. Then we discuss the implementation of the pre-processor in chapter 5 and the comparison process in chapter 6. Chapter 8 looks into the verification and validation of the heuristics. Chapter 9 focusses on the concludes and gives points for future work. The user guide for Holmes discussed in chapter 7.

# Chapter 2

# Context

## 2.1 Plagiarism

Plagiarism can be defined in many different ways. It is important to establish a definition of plagiarism and fraud. We need to know what to look for and what to do with it. In the Education and Examination Regulations[13] for our department, it says:

> Fraud and plagiarism are defined as actions, or failure to act, on the part of a student, as a result of which proper assessment of his/her knowledge, insight and skills, in full or in part, becomes impossible.

Copying other peoples work is considered plagiarism according to the definition. Courses and assignments are designed to develop and test some specific skills. If a student directly copies code for a program assignment he or she may not obtain the skills the assignment was designed for.

### 2.1.1 Reasons for plagiarism

There can be several reasons for committing plagiarism. For software plagiarism we can distinguish three different reasons.

1. Reason 1: lack of time
2. Reason 2: lack of programmer experience
3. Reason 3: a combination of Reason 1 and Reason 2

These three reasons are based on earlier findings in the course Imperative Programming[2]. Students who committed plagiarism and were caught with the

detection tool Marble[17] were interviewed about there motivation. We can conclude that students who plagiarise typically do not have the knowledge or the time to cover up there fraud. They are simply not able to change their submission beyond recognition. Changing the semantics of a submission costs too much time or is complicated. Often changing the semantics substantially is just as difficult, or even more difficult than program the assignment from scratch.

## 2.2 Detecting plagiarism

It is very difficult to conclude if a case of two programs that look similar is fraud or not. It can be a coincidence that two programs look similar, there is the possibility that students work together come up with a similar solution, and depending on the goal of the assignment, some cases can even be legitimate. For example, if the goal of an assignment is to see if a student can construct a particular program, it can be possible that a student changed an existing program so much that he has shown a level of understanding of the code that suits the goal of the assignment. In this case we can argue about the legitimacy.

Manual inspection by an assessor solves the judgement problem, but in a collection with $n$ students there are $\frac{n(n-1)}{2}$ possible cases of plagiarism. In Utrecht the $n$ often is larger then 100. This makes manual inspection infeasible.

Creating a tool for detecting plagiarism seems like a good idea, but as may be concluded from above, a tool can never offer conclusive evidence in all cases. The only thing a tool can do is measure the similarity level of two programs. The final decision must be made by the assessor. Therefore the goal of a plagiarism detection tool must be: to give an indication of plagiarism and to dismiss as many cases as possible where plagiarism is unlikely.

From experience, we know that the main reason for a student is to plagiarise:

**Observation 1** A student who plagiarises suffers from a lack of time, a lack of programmers experience or a combination of both.

A submission for a program assignment will be manually reviewed by an assessor because he has to grade the submission. This mains that the program must be understandable to the assessor. Using code obfuscators to prevent detection of plagiarism will not be useful for a student. Obvious transformations to hide plagiarism for a detection tool will also be noticed. For example: redefining the standard map function and call it applyOnAll just to prevent detection will be too obvious. (For those who are not familiar with the map function in Haskell. The map function applies a given function on all elements in a list and is defined in the Haskell Prelude) In short:

**Observation 2** A submission must be reasonable and human readable.

When a student commits plagiarism and tries to hide it, he may need to spend a lot of time on changing a submission. The changes must be substantial to avoid detection. Because of observation 1, a student will most likely focus on just one or a few methods of hiding plagiarism. Often a student attacks the plagiarism detector from only a single side. If the detector compares the program in many different ways some of the results will continue to score high. For example: if the student only translates the names of the identifiers and comments, the structure of the code is still the same. Concluding:

**Observation 3** A single hint of plagiarism is enough.

When there is a hint of plagiarism manual inspection is necessary. In our experience, manual inspection quickly tells us if plagiarism is the case or if there is another explanation for the level of similarity.

## 2.3 Refactoring

Refactoring[22, 20] is a technique for changing the structure of a program without changing the functionality of the program. The goal of refactoring is to improve the design and structure of the code. Refactoring is a technique that is often used in object-oriented languages to improve internal structure but it can also be used for changing the structure of functional programs.

The process of refactoring code has existed as long as programs have been written. In many programming languages it is obvious that a programmer can achieve a certain goal in many ways. Some of his options are better than others, but the result will be the same. When writing a program the code must be maintainable. That means that readability and reusability are very important. Refactoring is traditionally used to alter the code of a programmer to increase maintainability. The changes are purely structural and strictly separated from changes in functionality.

Refactoring a program can be done by hand or with some sort of tool. Nowadays there is tool support for many languages to make it easier to refactor the code. Regardless of the fact that it can be done with or without tools, refactoring can also be misused. When looking at refactoring from our point of view it can be used to camouflage plagiarism. If a student copies the code from another student and refactors it, it may not be that obvious anymore that it is actually plagiarism. If we take Haskell as an example then we can see that there are some simple things you can do to change only the structure of a program. You can do that without understanding what the code does. The most easy and commonly used technique is to give alternative names to definitions. By just changing some names, automatic detection or even detection by hand can be avoided. Another important thing to take into account is that some solutions can be written in

more than one way. Many languages including Haskell have multiple code constructions to express the same outcome. Another issue is that like in Java the order of the definitions in Haskell does not matter. The choice is up to the programmer: it depends on his style what structure he likes best. Nevertheless some of these structures can be easily changed to structures that will produce the same results, but look very different.

The presence of tool support only makes the problem bigger. If a tool can automatically refactor certain parts of code, then it typically costs less effort to do so. The reasons a student will commit plagiarism, as mentioned before, are lack of time and lack of knowledge. When a tool refactors the code, it will go much faster then by hand. It also doesn't require much knowledge on the part of the student because the refactoring is automatic.

On the next two pages some examples of refactoring can be found. Module $A$ is the original. In module $A'$ the identifier names have been changed by translating them from English to Dutch. Showing module $A$ and module $A'$ side by side it is obvious that the similarity level between the two is substantially high. The structure of the module and the structure of the individual function is exactly the same. In module $A''$ we performed multiple refactoring actions. Besides the name changing, we relocated every function and converted one function to a local definition. We also converted simple recursion to list comprehension in two functions and changed the pipe structure with only two option into a if-then-else construction. When comparing module $A$ with module $A''$, the similarity is far less obvious comparing to the earlier example. Moreover, we added two functions to module $A''$ that are never used and can be considered garbage. All these refactoring actions and the addition of unused code are quite easy to apply and prevent easy detection by manual inspection.

### module A

```
type Table = (String, [String], [[String
    ]])

tableWidth :: Table -> [Int]
tableWidth tabel = maxWidth (allWidth
    tabel)

allWidth :: Table -> [[Int]]
allWidth (_, b, c) = recordWidth c ++ [
    singleWidth b]

recordWidth :: [[String]] -> [[Int]]
recordWidth [] = []
recordWidth (x:xs) = [singleWidth x] ++
    recordWidth xs

singleWidth :: [String] -> [Int]
singleWidth [] = []
singleWidth (x:xs) = [length x] ++
    singleWidth xs

maxWidth :: [[Int]] -> [Int]
maxWidth [] = []
maxWidth (x:xs) = maxWidth' x xs

maxWidth' :: [Int] -> [[Int]] -> [Int]
maxWidth' [] _ = []
maxWidth' grootsten [] = grootsten
maxWidth' grootsten (x:xs) = maxWidth' (
    greatest grootsten x) xs

greatest :: [Int] -> [Int] -> [Int]
greatest [] [] = []
greatest [] (_:_) = []
greatest (_:_) [] = []
greatest (x:xs) (y:ys) | x >= y = [x] ++
    greatest xs ys
                       | otherwise = [y]
                           ++ greatest xs
                            ys
```

### module A'

```
type Matrix = (String, [String], [[String
    ]])

matrixBreedte :: Matrix -> [Int]
matrixBreedte tabel = maxBreedte (
    geheleBreedte tabel)

geheleBreedte :: Matrix -> [[Int]]
geheleBreedte (_, b, c) = registerBreedte
     c ++ [enkelBreedte b]

registerBreedte :: [[String]] -> [[Int]]
registerBreedte [] = []
registerBreedte (x:xs) = [enkelBreedte x]
     ++ registerBreedte xs

enkelBreedte :: [String] -> [Int]
enkelBreedte [] = []
enkelBreedte (x:xs) = [length x] ++
    enkelBreedte xs

maxBreedte :: [[Int]] -> [Int]
maxBreedte [] = []
maxBreedte (x:xs) = maxBreedte' x xs

maxBreedte' :: [Int] -> [[Int]] -> [Int]
maxBreedte' [] _ = []
maxBreedte' grootsten [] = grootsten
maxBreedte' grootsten (x:xs) = maxBreedte
    ' (grootste grootsten x) xs

grootste :: [Int] -> [Int] -> [Int]
grootste [] [] = []
grootste [] (_:_) = []
grootste (_:_) [] = []
grootste (x:xs) (y:ys) | x >= y = [x] ++
    grootste xs ys
                       | otherwise = [y]
                           ++ grootste xs
                            ys
```

### module A

```
type Table = (String, [String], [[String
    ]])

tableWidth :: Table -> [Int]
tableWidth tabel = maxWidth (allWidth
    tabel)

allWidth :: Table -> [[Int]]
allWidth (_, b, c) = recordWidth c ++ [
    singleWidth b]

recordWidth :: [[String]] -> [[Int]]
recordWidth [] = []
recordWidth (x:xs) = [singleWidth x] ++
    recordWidth xs

singleWidth :: [String] -> [Int]
singleWidth [] = []
singleWidth (x:xs) = [length x] ++
    singleWidth xs

maxWidth :: [[Int]] -> [Int]
maxWidth [] = []
maxWidth (x:xs) = maxWidth' x xs

maxWidth' :: [Int] -> [[Int]] -> [Int]
maxWidth' [] _ = []
maxWidth' grootsten [] = grootsten
maxWidth' grootsten (x:xs) = maxWidth' (
    greatest grootsten x) xs

greatest :: [Int] -> [Int] -> [Int]
greatest [] [] = []
greatest [] (_:_) = []
greatest (_:_) [] = []
greatest (x:xs) (y:ys) | x >= y = [x] ++
    greatest xs ys
                       | otherwise = [y]
                           ++ greatest xs
                            ys
```

### module A”

```
type Matrix = (String, [String], [[String
    ]])

registerBreedte :: [[String]] -> [[Int]]
registerBreedte lijst = [enkelBreedte x |
     x <- lijst]

breedteInfo :: [String] -> [(Int, String)
    ]
breedteInfo lijst = [(length x, x) | x <-
     lijst]

maxBreedte :: [[Int]] -> [Int]
maxBreedte [] = []
maxBreedte (x:xs) = maxBreedte' x xs
   where
    maxBreedte' :: [Int] -> [[Int]] -> [
        Int]
    maxBreedte' [] _ = []
    maxBreedte' grootsten [] = grootsten
    maxBreedte' grootsten (x:xs) =
        maxBreedte' (grootste grootsten x
        ) xs

geheleBreedte :: Matrix -> [[Int]]
geheleBreedte (_, b, c) = registerBreedte
     c ++ [enkelBreedte b]

grootste :: [Int] -> [Int] -> [Int]
grootste [] [] = []
grootste [] (_:_) = []
grootste (_:_) [] = []
grootste (x:xs) (y:ys) = if x >= y then [
    x] ++ grootste xs ys else [y] ++
    grootste xs ys

enkelBreedte :: [String] -> [Int]
enkelBreedte lijst = [length x | x <-
    lijst]

matrixBreedte :: Matrix -> [Int]
matrixBreedte tabel = maxBreedte (
    geheleBreedte tabel)

dubbelBreedte :: [String] -> [String] ->
    [Int]
dubbelBreedte lijst lijst2= [length x | x
     <- lijst]
```

## 2.4 Marble

Marble[17] is a plagiarism detection tool created by Jurriaan Hage at the Department of Information and Computing Sciences, Utrecht University. The program was developed because the creator wanted to know on what scale plagiarism was present within the department. The idea behind the program was to create a tool that will point out the existence of similarity between classes of Java-code. The pairs of classes that have the most similarities will end up at the top of the list. The user of Marble then has to inspect the code of each pair, starting at the top, and determine by hand if it is actually plagiarism. The detection of plagiarism in Marble is divided in two phases: normalisation and detection.

### 2.4.1 Normalisation

The normalisation phase prepares the source files of an assignment that is handed in. It brings the source files back to a list of tokens. The names of the character constants, hexadecimal numbers, numbers and identifiers are replaced by a single letter L,H,N or X. Important keywords, commas, brackets, important methods etc. are kept. All the tokens are saved in a file on a separate line. In Java, the order of method definitions is not important. Marble sorts the definitions in a special way to prevent that order of definition is a big issue.

### 2.4.2 Detection

In the detection phase, Marble uses the normalised versions to determine similarity. It compares all the assignments of the current incarnation, and it compares those assignments to the assignments in all previous incarnations. Marble uses a strict way to order the files and directories to make this possible. For comparing two normalised files, the standard Unix program diff is used. It compares two text based files with an algorithm that tries to find the largest common substring. The result will be the differences between the files. By calculating the length of the files and the number of lines they differ, a score can be generated. The score indicates how large the similarity between two files is. This way Marble can create a list of pairs with the highest similarity at the top and the pairs with the lowest similarity at the bottom.

### 2.4.3 Not sufficient

In practise it turns out that Marble works quite well on Java. With only one heuristic, structural comparison by lexical analysis, there are between five and fifteen cases of plagiarism detected every year in courses that use Java as the primary language. An experiment by applying a variant of Marble to Haskell programs wasn't successful. It seems that the single heuristic Marble uses is not

sufficient when comparing Haskell programs. The reason could be that Haskell has less structural information like the curly brackets and the semicolon than Java uses.

## 2.5 Moss

MOSS is a system created by UC Berkeley in 1994 [14]. MOSS stand for "Measurement of Software Similarity" and is able to detect plagiarism in many different programming languages. Widely used languages like C, C++, Java and C# are implemented in MOSS, but also functional languages like Haskell, ML and Lisp.

### 2.5.1 Winnowing

The technique MOSS relies on is called Winnowing[15]. The basic concept is to extract a fingerprint from a piece of text. The form of this fingerprint depends on the content of the text. By comparing fingerprint from different documents you can determine if parts of the documents are similar. The fingerprints for a document, in this case a piece of code, will be created by taking k-grams from the content. For example: the first two 4-grams of "haskelliscool" are "hask" and "aske". By hashing these k-grams and using the winnowing algorithm, a fingerprint will be created. The fingerprint will be a subset of all the created hashes from the k-grams.

Winnowing is actually a universal technique that can be applied to many different programming languages. As mentioned before, MOSS can be used for many known programming languages including Haskell. The problem we only face is that this universal technique doesn't take some of the research questions into account. Things like refactoring might be very hard to detect. In this case we know that we are dealing exclusively with Haskell, so we can use the language characteristics of this language to detect plagiarism.

## 2.6 Incarnations

It is likely that a certain assignment is given to students unchanged over a period of years. Therefore when comparing submissions on possible plagiarism, not only the submissions of that single incarnation need to be cross compared, also comparison between the incarnations is necessary. Moreover, our experience with Marble tells us that it is more likely that a student copies the work from someone in a previous year. A possible explanation is that there are students or former students who publish there work on the Internet.

It is essential that the detection of plagiarism is scalable. When a new batch of submissions arrives, these must be compared to each other and to all the

submission in previous incarnations. These older incarnations do not have to be cross-compared.

Marble already contains the feature of comparing incarnations as described above. When comparing Haskell submissions it is crucial that this way of comparing is preserved.

## 2.7 Templates

When source code is automatically compared, the number of false positives must be reduced to a minimum. We have to avoid comparing boilerplate code as much as possible because of the negative influence on the results. Something Marble doesn't support is the handling of code templates. A code template is a piece of code given to a student as part of an assignment. Their task is to modify or use the template code in a way specified by the assignment.

Due to these code template the level of similarity rises unintentionally. Since many of the assignments handed out in the functional programming course[1] are based on code templates, dealing with these templates is essential.

# Chapter 3

# Approach

Our approach in this project was to create a tool. In this tool we implemented all the ideas we had on how to detect plagiarism between Haskell submissions and applied it to actual Haskell programs.

## 3.1 Helium and AG

Before analysing source code we need to parse the Haskell code into abstract syntax. Our approach was to take Helium[8] and build our tool Holmes on top of the Helium parser. Helium is a compiler for a subset of the functional programming language Haskell. It is especially designed for teaching and learning Haskell. The compiler is developed and maintained at Utrecht University. The differences between Haskell and Helium language can be found on the Helium website [8].

The reason why we borrowed the parser from Helium is that Helium is built and maintained at the UU so it is easily available to us. Besides, we are familiar with the Helium system. The reason for developing a tool is to test our heuristics on Haskell programs. By using Helium we do not have to spend time on writing a parser but focus more the actual goal of this project. A parser transforms the literal code into abstract syntax. The abstract syntax in Holmes is described in the UUAG System [23] (Utrecht University Attribute Grammar System). Also a lot of the analysis in Helium is done with the UUAG. The choice for Helium gives us the advantage that we can reuse the UUAG code from Helium.

The UUAG System is an effective tool for abstract syntax tree computations, also known as syntax directed computations. The UUAG is used to implement attribute grammars. When working with a tree structure it is often needed to move data up or down the tree. If we want to know the minimum value of all

leafs in figure 3.1, the values of all the leafs need to be transfered up the tree to compare them at the top Bin node. The other way around, transferring data down the tree is also often necessary. In the UUAG System this can be done by creating synthesised and inherited attributes in a pretty straightforward way. For more information about the UUAG System, see [23, 21]

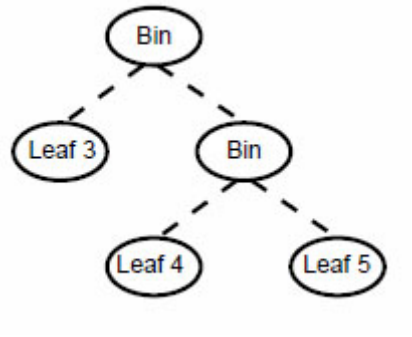


Figure 3.1: Simple tree structure

## 3.2 Heuristics

According to observation 3 in section 2.2 we only need one hint of plagiarism. This follows from observation 1 that most students will tend to attack a detection mechanism from one or a few angles. Therefore our approach is to simply come up with a broad selection of heuristics to measure similarity in Haskell programs. By implementing these heuristics in a tool and running this tool, it is possible to decide which heuristics are most useful.

To bring some structure in the heuristics we classify them into three categories: structural heuristics, literal heuristics and semantic heuristics.

### 3.2.1 Structural heuristics

The structural heuristics obviously focus on the structure of the program. We compare two different aspect of the structure for now.

The first thing we compare is the source itself, similar to Marble. This is done by performing **lexical analysis**. Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. All the important information is represented by tokens and all the non-important information is eliminated. By defining a correct procedure for tokenizing the Haskell source, we are able to compare the important parts of programs by comparing the token streams.

The second aspect of structural comparison is comparing the **number of parameters** for functions. It is hard to change the number of parameters for a

substantial amount of functions without notice. According to observation 2 this is likely to be noticed during manual grading.

### 3.2.2 Literal heuristics

The literal heuristics in this project focus on text. Literal text refers to the text that is placed directly in the code. We distinguish two types of literal text: **comments** and **literal strings**. When extracting and comparing both the collection of strings and the collection of comments, we determine the literal similarity. Although we know that strings and comments are easily changeable, it is generally very suspicious when the level of similarity for this type of heuristic is substantially high.

### 3.2.3 Semantic heuristics

To compare the semantics of programs we choose to compare the graphs that represent the calls between functions in a program, the **call graph**.

A *graph* [16] is an abstract representation of a set of objects and the relations between them. The objects are usually called *vertices* or *nodes*. A link between two nodes is called an *edge*. More formal:

**graph** A graph is a pair $G = (V, E)$ of sets satisfying $E \subseteq V \times V$ ; thus, the elements of $E$ are 2-element subsets of $V$. The elements of $V$ are the *vertices*(or *nodes*) of the graph $G$, the elements of $E$ are its *edges*

A call graph is a *directed graph* that represents the relation between functions or subroutines in a computer program. Every edge is not only a link between the functions, but it also has a direction.

**directed graph** A directed graph (or digraph) is the same as a normal graph with only one difference. The elements of $E$ or not 2-element subsets but **pairs** of $V$

In a call graph the functions are represented by the nodes. The directed edges between them describe the call relations. There is an edge from node X to node Y when the code for X includes a call to Y. For example, figure 3.2 is the call graph representation of the code below.

```
functionA :: String
functionA = functionB "foo" "bar"

functionB :: String -> String -> String
functionB str1 str2 = str1 ++ str2
```

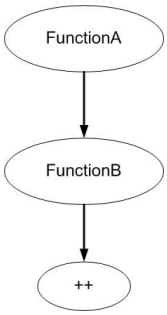


Figure 3.2: Call graph example

By comparing the call graph we can do analysis on the submission as a whole regardless of the fact that the program code may be split over multiple files. When comparing call graphs, we need to find to level of similarity between them. When the structure of two call graphs are identical regardless of the labelling and layout the graphs are *isomorphic* (see figure 3.3). Testing for isomorphism is easy but slow and not sufficient. A minor change in the structure of one of the graphs means that the graphs are still quite similar but not ismorph. At this point there is no available algorithm to calculate approximate isomorphism. Therefore we compute a metric for each graph, to expresses the similarity between these graphs and compare graphs by comparing the metrics.

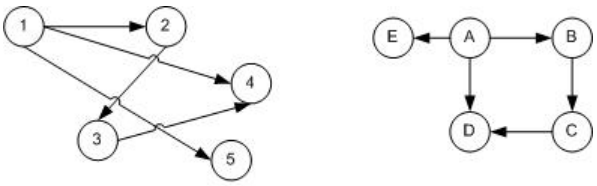


Figure 3.3: Graph isomorphism

### 3.2.4 Fingerprints

Besides using our own defined heuristics we also use fingerprinting and winnowing. MOSS [14] uses these techniques and has proven to be effective in many cases. The reason to implement fingerprinting in Holmes is to compare the results of the other heuristics against it and to see how well it does when template and dead code removal are used to pre-process the submissions.

## 3.3 Sensitivity Analysis

To see if the heuristics work, we have to devise a suitable set of tests. In sensitivity analysis we measure how the heuristics act to changes in the program. By taking a Helium compatible file and perform all sorts of (refactoring) actions on it to hide similarity, we can verify that the heuristics behave as expected. We also create programs that combine a number of refactoring actions and compare them against the original. This way we check how sensitive a heuristic is when multiple refectorings have taken place.

## 3.4 Validation

Besides sensitivity analysis, it is important to see how the heuristics do on actual submissions. To test and validate the heuristics we apply Holmes on a number of collections of submissions. By using the submissions from the Functional Programming course we have real data to validate the heuristics with. The importance to test Holmes on real data is that pre-fabricated test material only tend to test some particular issues. With real data we don't know what results to expect. We choose the data from the Functional Programming course because it is available to us and holds a substantial amount of submissions that can be used for this validation. The submission were written by students without knowledge about any plagiarism detection system.

# Chapter 4

# Architecture

As with Marble [17], plagiarism detection with Holmes is divided into two steps. At first we have to prepare the submissions for comparison. We extract the relevant data and store this for later use. This part is called the pre-processing phase. The second phase is the comparison phase where the submissions are compared based on the data extracted by the pre-processor.

## 4.1 Pre-processing

The pre-processor takes a submission and performs the following main tasks

i. Remove all the information that is not useful for, or interferes with the comparison.

ii. Pre-calculate and store all important information on disk.

Figure 4.1 shows the architectural overview of the pre-processor. The ellipses represent the input and output of the system. The boxes represent the subsystems of the pre-processor.

When normalising the program to an abstraction that only contains functions that are important for comparison, two things need to be calculated. We need to know which functions are part of the template code (see section 2.7) and we need to calculate the call graph (section 3.2.1) to see if there are unused functions. With the outcome of both calculations together we can distinguish between useful and irrelevant functions and remove the latter. We refer to this as the *abstraction*

The abstraction is used to calculate all the important program information which is stored to disk. Note that extraction of comments is done before the abstraction is created. The Helium front-end doesn't support comments; therefore we created a separate parser that can handle comments. The advantage is that

we didn't have to edit the current lexer, parser and abstract syntax declaration which saved a lot of time and makes integration of a newer version of Helium easier.

All the data necessary for comparison is extracted and saved separate from the original source. Once this data is stored it can be used multiple times, so pre-processing has to be done only once.

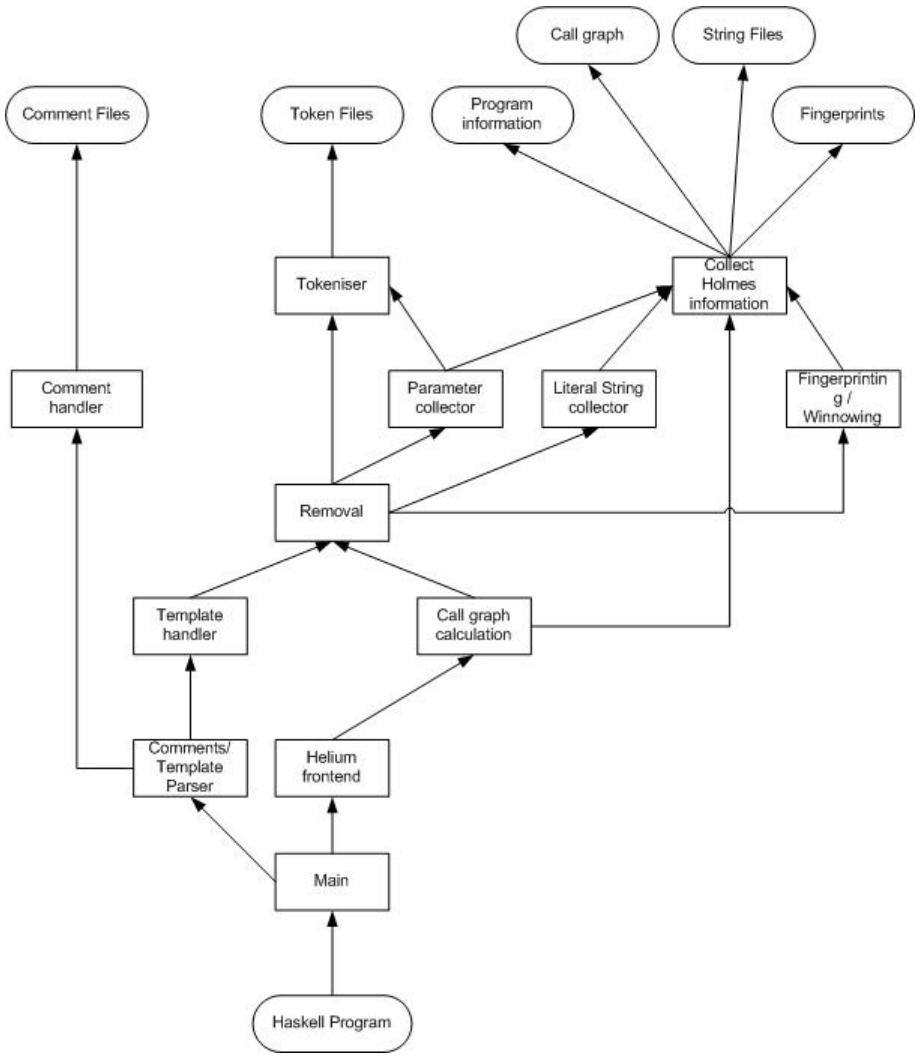


Figure 4.1: Architectural overview pre-processor

## 4.2 Comparison

The main task of the comparison tool is to read all the information that was saved by the pre-processor and compare it with the same information for other programs. For a complete comparison, it has to compare all the submissions

within one incarnation and also compare it with the submissions from previous incarnations. For scalability it is important that all these previous incarnations are not again compared to each other. Therefore a strict separation between the new and the older incarnations is required. The comparison of submissions is done on two different levels.

**Submission level** Some of the information collect by the pre-processor can only be used if the submission as a whole is compared. All the information about the call graph and the call graph itself is an example of information that is only valid for a complete submission. The information calculated per file is already, or will be, integrated into a complete submission. Comparing a submission as a whole means that this way of comparing, when correctly done, is file independent. It does not matter if somebody put a complete program in a single file or spread the code over multiple files.

For the submission based comparison we compare the call graph and all the information generated from the call graph, the unified token stream, unified string files, unified comment files and the function parameters.

**File level** Besides comparison at submission level it is still necessary to compare single modules in a submission. It is possible that a student just copied a single library or utility file. When the size of a program is substantial this may be unnoticed during submission comparison. It could be even worse when someone partially copies a file and appends self written code. Not all forms of comparison used to compare a complete submission are suitable for comparing individual modules. Therefore we only compare the individual tokens stream, literal strings and comments for file to file comparison.

# Chapter 5

# The Pre-Processor

The pre-processor is the first part of the Holmes system. Every submission that needs to be checked for plagiarism with Holmes must be prepared (*normalized*) by the pre-processor. **Sherlock**, as the pre-processor is called, normalizes the source code by extracting only the important information needed to detect plagiarism and removes the information that may interfere when comparing different submissions.

Sherlock is based on the Helium[8] compiler, see section 3.1. To normalize the source code parsing the Haskell code was unavoidable. By using Helium we not only avoided the work of writing a new parser for Haskell, we also used part of the existing analyses implemented in Helium. In fact the compilation process in Helium contains 10 phases. By using the first four phases in Helium: *lexing, parsing, importing* and *resolving operators*, we can transform a program to abstract syntax in such a way that have we enough information for the normalization process. On top of this stripped version of Helium we implemented several analyses to compute the normalized information per module / submission in Haskell programs.

Normally Helium applies all ten phases recursively over the imports. In phase 10 a so called lvm file will be constructed and saved; lvm files are the object / class files of Helium. When a submission contains multiple files, phases 3 (importing) needs the lvm file of the imported module(s) to compile. For Sherlock we didn't need all ten phases but we do need the lvm file. Therefore we decided to force recompilation by Helium to ensure that the lvm files are up to date. The submissions must be Helium compatible anyway and changing the current structure to avoid total recompilation would be a waste of time considering the objective of this thesis.

In appendix F a simple demonstration program with the corresponding output is published. This is used to illustrate the work and output of the pre-processor.

## 5.1 Abstraction

The *abstraction* is the useful source code without irrelevant function as explained in section 4.1. An abstraction is only computed for the useful part of the call graph. For this abstraction we simple remove all functions not present in the abstract call graph and we omit the functions that are part of the template code. The abstract call graph is created using the declared start points in the config file (see 7.4.1). All functions reachable from these start points are considered useful. Note that the more specific the declared start points are, the more useful the abstract call graph is. We also decided that the abstraction will only contain functions; data definitions and import declarations are omitted. The functionality of a program in functional language is specified in the functions so we believe comparing the functions is sufficient.

When looking at the source of appendix F, knowing that the only start point of this program is Demo.printTable, the abstraction will contain all functions except Demo.top10.

## 5.2 Templates

To obtain only the useful functions, we want to omit template code. We decided to create an annotation especially for templates. This annotation should not interfere with any compiler or alter the original Haskell code which is why pragma's are used [11].

Pragma's are used to give additional instructions or hints to the compiler. It could affect the efficiency for example. A normal pragma has the form {`-# type ... #-`} where type indicates the type of pragma. For a normal compiler a pragma looks like ordinary comments which means that the Haskell code stays untouched. However, it is not convenient to expand the pragma language with a pragma just for Holmes. Therefore we only took the idea of the pragma and created our own version, the HolmesPragma. Just like the normal pragma the HolmesPragma will have the appearance of a comment block. The form of the HolmesPragma will be like {`-H ... H-`}. At this time only template code will be annotated by a HolmesPragma, but the pragma can be expanded for other uses within the Holmes system.

Template handling in Sherlock is done on the function level. It is possible to divide the functions in several different template categories: code that may not be changed, code that may be changed, code that must be changed etc. All the categories describe a level of possible similarity. We decided to use only two categories:

- The function MUST be removed before comparison
- The function MUST NOT be removed before comparison

The creator of the template should decide which functions are useful for comparison. It is clear that predefined function where the code may not change need to be marked template. Nevertheless, there may be functions where the code can be changed but the function is still template. An example is a pretty printer: although the implementation may be different between students, it may not be part of the real functionality of the program. In that case the pretty printer can be marked as template. We decided that all functions marked template will be removed. A function is either useful for comparison or not; the decision is up to the creator.

## 5.3 Heuristics

To compare modules and complete submissions we implemented a variety of heuristics. Part of the normalisation process is to compute the values for all these different heuristics. We divided the heuristics in three main categories: structural, literal and semantic heuristics. Per category we discuss how we implemented the particular heuristic.

### 5.3.1 Structural heuristics

#### Tokenizing

We tokenize the source code to make sure that easily transformable details that do not interfere with the meaning of the program are removed. However, we want to keep language specific symbols like the double colons, arrows, curly brackets, round brackets etc. White spaces will be removed and literals and identifiers will be replaced by a symbol. The literal integers, characters, floating points and strings will respectively be replaced by the characters I, C, F and S. Operators will be replaced by the character O and the identifier names will be represented by the character X. For the identifiers there is an exception list containing widely used identifiers like `Int, String, Bool, Maybe, Either, Show, Eq` etc. The initial set of exceptions are all types (with there constructors) and classes defined in the Prelude. These identifier names will not be replaced. It is possible to alter the set of exceptions if necessary.

All these actions transform a Haskell source file to list of tokens. These token lists will be saved in two ways, sorted and unsorted. In Haskell it does not matter in what order top level functions are declared. This means that the order is easily changeable. To be independent of this choice we order the token stream. Similar to Marble, the sorted and unsorted token streams will be both saved to disc separately.

For the sorted output of a module, the functions in the abstraction are sorted before feeding it to the tokenizer. The deterministic ordering of the tokens is

based on three aspects.

i. The number of parameters

ii. The number of tokens (essentially the size of the function)

iii. Alphabetic order of the concatenated list of tokens of that particular function

If two functions still have the same result, we consider them identical. In that case the order between those two function is not relevant.

The implementation of the tokenizer is based on the existing pretty printer. By replacing the literals with the corresponding tokens we are able to print out the tokens in the right layout. A list of special identifiers is declared containing all the identifiers which we want to keep. So before an identifier will be printed it is compared against this exception list. Because the tokenizer is based on the pretty printer the layout is preserved and all the special characters, e.g., function arrows, are inserted in the output. This prevents silly mistakes and it is easy to conclude if the output is correct and satisfying.

The tokens are saved so that every token is printed on a separate line. The sorted and unsorted tokens are saved in separate files with respectively the `.tok` and `.tks` extension for every Haskell module. For the complete submission only the sorted tokens are saved in a file called `all.tks`. Saving the unsorted tokens for a complete submission would be useless.

In appendix F.2, both token outputs of the demonstration code are published. We can clearly see the difference. The unsorted token stream (`Demo.hs.tok`) starts with the token presentation of the function `printTable`. The sorted stream (`Demo.hs.tks`) however, starts with the token presentation of the function `fit` because this function has the largest amount of parameters.

### Parameters

The number of parameters is not only used for direct comparison but as described in section 5.3.1 it is also used for sorting the token stream. Therefore not only the number of parameters is important but also the corresponding name of the function.

According to the abstract syntax we use from Helium, only a `FunctionBinding` contains potential parameters. With the UUAG we declared a inherited attribute to collect the parameters. In the code snippet below you can see the rules that describe how to obtain the parameters.

```
...

SEM FunctionBindings
    | Cons                          lhs.parameters = @hd.parameters
```

```
    | Nil                              lhs.parameters = error "Error:
        Collecting parameters failed"

SEM FunctionBinding
    | FunctionBinding                  lhs.parameters = @lefthandside.
        parameters

SEM LeftHandSide
    | Function                         lhs.parameters = (@name.self,
        length @patterns.self)
    | Infix                            lhs.parameters = (@operator.self
        ,2)
    | Parenthesized                    lhs.parameters = @lefthandside.
        parameters
```

In short, for every `FunctionBinding` we want the parameters from the `lefthandside`. For every alternative of `LeftHandSide` we describe how to create the tuple that contains the name of the function and the number of parameters. All elements in `FunctionBindings` apply to the same function, so the number of parameters is equal for every element. Therefore we only need the parameters of the first item.

The list containing the number of parameters per function will be saved to disc without the names of the functions. The name of a function in this case is only important for the sort strategy of the token stream. When comparing the number of parameters directly the function names, also called identifiers, are not used due to the fact that identifiers are easily changeable.

The parameters are saved, among other value, in the file `metadata.hol`. An example can be found in appendix F.3.

### 5.3.2 Literal Heuristics

#### Strings

Collecting the literal strings in the source is quite easy: the Helium parser collects the strings for us. In the Helium abstract syntax a literal string is part of the `Literal` data type. Using the UUAG we declared an inherited attribute to collect the strings. We are not interested in the values of the other alternatives of `Literal`.

```
SEM Literal
    | Int       lhs.string = []
    | Char      lhs.string = []
    | Float     lhs.string = []
    | String    lhs.string = [@value]
```

Only the literal strings from the abstractions are collected. This way literal strings from unused functions are not included. For every Haskell file, the collected literal strings are saved in a file `<originalname>.str`. Every sting is

splitted into single words. These words are converted to lower case and filtered on non-alpha characters Numeric characters, special characters and white spaces will be omitted to prevent overhead in the string output. The words are first sorted on length then on alfabetic order and each printed on a separate line in the file `<originalname>.str`.

There is no example of a string file in appendix F. The explanation is that the string file created for the demonstration code is empty. The only literal strings are declared in the function `Demo.top10`. Since this function is not part of the abstraction, no literal string are saved. All other literal strings used to print the table are filtered because the individual characters do not belong to the alpha character set.

#### Comments

The Helium lexer omits comments. There is also no support for comments in the Helium parser and the abstract syntax. We chose to create a separate lexer and transform the comment syntax directly into abstract syntax. The reason for this separate process is the complexity of the existing parsing process. Implementing the handling of comments in the Helium front-end means changing the lexer, the parser and the abstract syntax. Creating a separate process for handling comments not only saved us time but the integration of a modified Helium front-end will be easier.

The downside is that comments related to unused functions are accepted as well. On the other hand, we cannot automatically determine if a piece of comment is related to a particular function so we might as well include them all. The comments will be saved to the file `<originalname>.comm` in a similar format as used for the literal strings. An example of a comment file can be found in appendix F.4.

### 5.3.3 Semantic heuristics

#### Callgraph

Before we can compare the call graph or in our case values that express the similarity between them, we have to construct the call graph first. The call graph in Holmes is represented by a list of tuples. Every tuple contains the name of a function and a list of names for the top-level functions it calls. So for every declared top-level function we basically collect the function calls.

```
SEM Declaration
  | FunctionBindings lhs . calls = @bindings.calls
  | PatternBinding   lhs . calls = @righthandside.calls
```

But when collecting the calls we have to take care of several things. In our case the call graph only displays the relation between functions on top level.

Local definitions in a function when using **where** or **let** do not end up in the call graph. For expressions with more then one possible outcome, as the conditional expressions, we take the calls from the guard and all possible outcome together.

```
SEM RightHandSide
  | Expression  lhs . calls = solveLocalScope @expression.calls
      @where.transitions
  | Guarded lhs . calls = solveLocalScope @guardedexpressions.calls
      @where.transitions

{
solveLocalScope :: Names -> Names -> NameNamesTups -> Names
solveLocalScope _ [] _ = []
solveLocalScope done nms trans = if newScope == scope then scope else
     solveLocalScope newDone newScope trans
        where
                scope = sort $ nub nms
                toReplace = scope \\ done
                newDone = done ++ toReplace
                newScope = sort $ nub $ (scope \\ toReplace) ++
                    concatMap (flip lookupNNTs trans) toReplace



lookupNNTs :: Name -> NameNamesTups -> Names
lookupNNTs name nnts = if replacement == [] then [name] else
    replacement
        where
                replacement = concat [calls | (nm, calls) <- nnts, nm
                    == name]
--$
}
```

To avoid calls to local definitions in the final call graph we take the calls from a function and replace the local definitions with its calls in an iterative process. If a function calls a locally defined function, the call will be replaced by the calls from the local definition. By iterating this process until the outcome stabilises gives us only the calls to top level functions. Note that a definition will only be replaced once (the replaced definitions will be saved) to prevent infinite loops caused by recursion

We take the function `writeTable` from appendix F as example to illustrate this iterative process.

| Solve local scope function writeTable (appendix F) | | |
|---|---|---|
| local definitions = (maxWidth,[columnWidth]),(writeName,[++]), (writeRecord,[++,fit,maxwidth]), (writeLine,[replicate, foldr, +, maxwidth,++]) | | |
| Iteration | Scope | Done Defenitions |
| Init | [writeName, writeRecord writeLine, ++, concatmap] | [] |
| 1 | [++, maxwidth, replicate, foldr, +, concatMap] | [writeName, writeRecord writeLine, ++, concatmap] |
| 2 | [++, replicate, foldr, +, concatMap, columnWidth] | [writeName, writeRecord writeLine, ++, concatmap, maxwidth, replicate, foldr, +] |
| 3 | [++, replicate, foldr, +, concatMap, columnWidth] | [writeName, writeRecord writeLine, ++, concatmap, maxwidth, replicate, foldr, +, columnWidth] |

In every iteration we substitute all possible definitions in the scope with the calls in the particular local definition. All definitions that can not be replaced will stay in scope. When substituting a definition we first filter the calls that are already replaced. The results in iteration 3 are equal to the results of iteration 2, so the outcome is stable. We now know all outgoing top level function.

By putting the function name and the calls together in a tuple we basically have all function calls per function. For the `FunctionBindings` we only need one of the entries, because all items in the list are related to the same function.

**Full qualification** When we have all the function calls per function for every module we need to make the names fully qualified. Helium does not support fully qualified names because Helium exports everything it imports. This means that if `module A` imports `module B` and `module B` imports `module C`, all functions from `module C` are available in `module A`. Moreover when `module A` uses a function imported from `module B` it can originate in either `module B` or `module C` doesn't know that the function are declared in `module C` because it imports all functions from `module B`.

To create fully qualified names we added an option to Helium. With the *–H-fullqualification* option Helium now outputs an **.fqn** file for every module. This file contains the local scope of the particular module. When mapping the list of function calls per function interactively over the local scopes we end up with actual fully qualified name. This is only possible because in Helium a function name can only be declared once, because of the import and export strategy Helium uses.

**Abstract call graph** The list with calls per function is a representation of the complete call graph of the program. However we are not interested in the complete call graph. We only want the call graph of the part that may be accessed. In other words we do not want to have dead or garbage functions in our call graph. Therefore the user can define one or multiple start points(see section 7.4.1). Only functions reachable from these start points are considered useful.

**Expressing call graphs** To calculate the similarity between call graphs we need a way to express them. At this time there is very little known about computing approximate isomorphism on call graphs. What we do know is that the complexity of such an operation will be high. Therefore to express a call graph we started to calculate the degrees: the in-degree, the out-degree and the total-degree. The in-degree for a single node is the number of incoming edges. Logically the out-degree concerns the outgoing edges and the total-degree is the sum of incoming and outgoing edges. More formally:

Let $G = (V, E)$ be a (non-empty) directed graph where $E \subseteq V \times V$.
$in - N_G(v) = \{w|(w, v) \in E\}$
$out - N_G(v) = \{w|(v, w) \in E\}$
$N_G(v) = in - N_G(v) \cup out - N_G(v)$

**total-degree** The total-degree $td_G(v)$ is the cardinality of the neighbours $td_G(v) = |N_G(v)|$.

**in-degree** The in-degree $id_G(v)$ is the cardinality of the incoming neighbours $id_G(v) = |in - N_G(v)|$

**out-degree** The out-degree $od_G(v)$ is the cardinality of the outgoing neighbours $od_G(v) = |out - N_G(v)|$

For every type of degree we create a sorted list of the number of edges per node. We also calculate the minimum, maximum and average per degree. Besides the degree characteristics we also calculate the diameter of a graph. The diameter is defined as the longest shortest path between any two nodes. Simply calculating the shortest paths between all possible nodes and selecting the longest path is sufficient. .

Let $d(v)$ be one of the degrees $id(v)$, $od(v)$ or $td(v)$ of a vertex $(v)$

**minimum degree** The minimum degree of $(G)$ is $\delta_d(G) := min\{d(v)|v \in V\}$

**maximum degree** The maximum degree of $(G)$ is $\Delta_d(G) := max\{d(v)|v \in V\}$

**average degree** The number $\bar{x}_d(G) := \frac{1}{|V|} \sum_{v \in V} d(v)$ is the average degree

**diameter** The distance $dis_G(x, y)$ in $G$ of two vertices $x, y$ is the length of a shortest path from $x$ to $y$ in $G$; if no such path exist $dis(x, y) := \infty$. The diameter of $G$ is $diam(G) := max\{dis(x, y)|x, y \in V, x \neq y\}$

All values derived from the graph and the graph itself will be stored. Storage of the call graph itself is in the DOT [3] format. We generate a dot file in which the nodes and edges are defined separately according to the DOT specifications.

The DOT specification is chosen because of the simple structure. Besides if the drawing tool *dot* [5] is installed it is possible to create a visual representation of the call graph. The pre-process will output this visualisation as a PNG image. The rest of the information calculated from the call graph will be stored as text in a separate file for meta data, `metadata.hol`. In appendix F.5 a complete dot specification and the visual call graph are published. The meta data file can be found in F.3

## 5.4 Fingerprints

The idea of creating document fingerprints is not new [15]. To use it on Haskell we need to set the proper k-gram and window sizes. Because MOSS [14] supports Haskell programs we contacted Alex Aiken, creator of MOSS, with these questions. His answer was:

> *I've generally found that a k-gram value of between 25 and 30 works well. As for window size it depends entirely on the size of the programs being compared. For a small batch a window size of 5 gives good performance; in runs comparing all of the open source code I could find on the internet I've used window sizes of 100.*

With this advice the implementation of fingerprinting in Holmes was done with a k-gram value of 25. A window size of 5 was chosen because in Holmes the fingerprinting is part of the pre-processing. We do not know beforehand what the size of the batch is that we need to compare. The suggestion to increase the window size when the batch increases has two reasons: make it feasible because more hashes mean more time. The second reason may be that with a greater window size there wil be fewer false positives. Part of the feasibility is solved because we compute the fingerprint just once in the pre-phase. Besides that, MOSS cross compares all entities in a batch, so feasibility is a bigger issue. Holmes on the other hand will just the compare new programs to the already existing programs (and to each other).

## 5.5 Output

The pre-processor outputs all the data to files located in the root of the submission. The files are arranged in the following way

| | |
|---|---|
| directory **tokens** | |
| *orginalname*.tok | unsorted tokens stream for a module |
| *orginalname*.tks | sorted tokens stream for a module |
| all.tks | sorted tokens for the complete submission |
| directory **StringComments** | |
| *orginalname*.str | literal strings in a module |
| *orginalname*.comm | comments in a module |
| directory **fingerprints** | |
| *orginalname*.fpr | fingerprint output for a module |
| all.fpr | fingerprint output for the complete submission |
| directory **holmesdata** | |
| callgraph.dot | the dot specification of the call graph |
| callgraph.png | the visual representation of the call graph |
| metadata.hol | file with meta data containing:<br>- all three sorted degree lists<br>- the minimum, maximum and average of every degree list<br>- the diameter of the call graph<br>- the sorted list containing the number of parameters for each top-level function |

# Chapter 6

# Comparing

The compare tool is the second part of the Holmes system. For comparing the submissions we use the data produced by the pre-processor. Comparison in Holmes (as the compare tool is called) is done on two different levels: submission level and file level. For comparison on submission level we take the submission as a whole and compare it with the other submissions. Comparison on file level means that every file will individually be cross compared to all files of the other submissions.

Some of the heuristics can only be applied in a comparison on submission level. The comparison of the call graph characteristics is an example. For the heuristics applied on both submission level and file level comparison the exact same technique will be used to calculate a score. The score for every heuristic is between 0 and 100 where 100 represents identical and 0 means totally different.

## 6.1 Reading files

For every submission we need to read several files. Because file IO is an expensive operation this is only done once. The data is parsed and saved in special datatype.

```
data ProgData = ProgData FilePath TokenData TokenData StringData
    CommentData FingerPrintData MetaListData MetaValueData deriving
    Show
```

Because of lazy evaluation in Haskell only the files that are really needed will be read. For example when only submission level comparison is performed, the files used for file level comparison will not be read.

## 6.2 Semantic heuristics

### 6.2.1 Call graph

We compare the call graph by individually comparing all items extracted from the call graph by the pre-processor. This heuristic can only be applied on submission level because the call graph can only be created for the submission as a whole.

#### Degree lists

The first step is comparing the three different degree lists (indegree, outdegree and totaldegree). We formulated three different ways to calculate a score for the similarity level of two degree lists. All three algorithms will be applied to all three different degree lists.

**algorithm 1** By calculating the edit distance we can determine a certain level of similarity. The Levenshstein distance [9] is a metric for measuring the amount of difference between two sequences. Usually this is applied to strings to calculate the so called edit distance. The Levenshtein distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. By implementing the Levenshstein distance for lists, it can be used on Integer lists like the degree lists. For implementing the Levenshstein distance we choose a dynamic programming approach. The maximum distance is equal to the length of the longest list so:

```
maxlength = max (length list1) (length list2)
distance = levenshsteinDistance list1 list2
score = 100 * (maxlength - distance) / maxlength
```

**algorithm 2** In algorithm 1, when substituting an item, the difference between these items is not of importance. This means that the difference between [1,2,3] and [1,2,6] is equal to the difference between [1,2,3] and [1,2,88]. In our opinion the difference between these two pairs is substantial. In this algorithm we transform the degree list. Not the numbers in the list, but the position will displayed. The numbers represent the amount of time the position has to be printed. For example: [1,2,3] will be [0,1,1,2,2,2].

| position | 0 | 1 | 2 |
|---|---|---|---|
| value | 1 | 2 | 3 |
| outcome | 0 | 11 | 222 |

Now when applying the Levenshstein distance over the transformed degree list, the value of a degree will also play a role. When we now compare [1,2,3] and [1,2,6] we calculate the Levenshstein distance between [0,1,1,2,2,2] and

[0,1,1,2,2,2,2,2,2]. The outcome is 3 instead of 1 (algortihm 1). The larger difference between the values of the degree the higher the Levenshstein Distance will be with this algorithm.

**algorithm 3** When comparing [1,2,4] and [1,2,6], the list [1,2] is equal which is $\frac{2}{3}$ of the list. This leaves us with 4 and 6 which is $\frac{1}{3}$. When looking at the numbers 4 and 6 we can conclude that both nodes have 4 edges. So, $\frac{4}{6}$ is equal and $\frac{2}{6}$ is different. Therefore the similarity is $\frac{2}{3} + (\frac{4}{6} * \frac{1}{3}) = 0.88888$ which makes the score 88.888

When comparing lists of different sizes the same principle holds. For example when comparing [1,2,2,5,6] and [1,2,3]: The numbers [1,2] are equal which is $\frac{2}{5}$. The difference is [2,5,6] and [3]. We compare the 3 with the closest number in the other list. In this case 2. That means for the comparison of those particular nodes that $\frac{2}{3}$ is equal and $\frac{1}{3}$ is different. Concluding, the similarity is $\frac{2}{5} + (\frac{2}{3} * \frac{1}{5}) = 0.533333$, meaning a score of 53.333

#### Derived values

Not only the degree lists but also the single values derived from the call graph are compared. These values are the **minimum**, **maximum** and **average** of every degree list and the **diameter** of the call graph. For the comparison of these individual values we also present the outcome as a number between 0 and 100 using the function `compareSingle`:

```
compareSingle :: Double -> Double -> Double
compareSingle a b = 100 * ((biggest - (abs (a-b))) / biggest)
        where
                biggest = max a b
```

## 6.3 Structural heuristics

### 6.3.1 Token comparison

The comparison of tokens streams is an example of a heuristic that is applied both at file level and submission level. For the file level comparison both the sorted and the unsorted token streams will be compared. For comparison on submission level only the prepared sorted token stream (saved in all.tks) will be compared.

For all alternatives we use the same method to calculate the score that describes the similarity between the two streams. We use the diff library for Haskell [7] to calculate the difference between the two token streams. Our first alternative was, similar to Marble, to use Unix diff. In Holmes however we need to process the score afterwards. It is possible to import the score from Unix diff back into Holmes but it is not convenient. Because the diff library was available to us, we

decided to use this option. Another important advantage is that we now decrease the number of IO operations with this option. These operations tend to be time consuming. The algorithm to calculate the final score is :

```
totallength = (length file1) + (length file2)
tokendiff = diff file1 file2
similarity = 100 * (totallength - tokendiff) / totallength
```

### 6.3.2 Parameters

The arity of all functions in a submission is represented as a sorted list of integers. Every integer represents the arity for a particular function. This representation is the same as the representation of the degree lists from a call graph. Therefore we use the same algorithms to express the similarity between the parameters per function for a complete submission (see section: 6.2.1).

## 6.4 Literal Heuristics

Comments and strings are compared in the same way. The output of those two look like the output of a token stream. Only we cannot use the compare algorithm for tokens on these literals. A literal in this case can contain more than one item. It can be a sentence for example. This means we also need to check on approximate string similarity.

We first implemented a version that uses the Levenshtein distance [9] to calculate the edit distance for every possible string couple from two sources. In this version the strings were not split into separate words by the pre-processor. This implementation turned out to be extremely time consuming. Therefore we chose the option to change the pre-processor output as described in section 5.3.2 and use the exact same algorithm as for comparing token streams. To illustrate this:

When having the following two set of strings

| | |
|---|---|
| ”lorum ipsum” | ”dolor eu” |
| ”dolor sit amet” | ”lorum ipbus” |
| ”consectetur” | ”exesectetion” |

In the first implementation we calculated the Levenshtein distance for all **9 possible couples** to find the closest match for every string. The Levenshtein distance is a very time consuming operation, especially when the sets are large. Therefore we in the current implementation we transform the strings into:

| | |
|---|---|
| ”sit” | ”eu” |
| ”amet” | ”dolor” |
| ”dolor” | ”lorum” |
| ”ipsum” | ”ipsub” |
| ”lorum” | ”exesectetion” |
| ”consectetur” | |

In the current implementation we transform the sets into a sorted set of single words. We sort on length followed by alphabetic order. Similar to the token comparison we now apply a diff on the set. This single diff operations saves a lot of time comparing to the calculation of all the Levenshtein distances.

# Chapter 7

# Using Holmes

## 7.1 Helium

Holmes uses Helium[8]. Therefore you must be certain that a proper Helium version is installed on your system. When installing a Helium version make sure you use the right source by choosing either of the following option:

- a helium system source repository from the website dated 15 jan 2010 or later
- the helium svn repository (https://subversion.cs.uu.nl/repos/staff.jur.heliumsystem/README)

## 7.2 Compilation

Holmes is based on the Helium compiler. Therefore there are some dependencies to other parts like LVM[19, 10] and TOP[18, 12]. The Holmes package contains two programs:

- The pre-processor (preparation / normalisation program) called **Sherlock**
- The compare program called **Holmes**

After compilation, both programs can be found in the directory helium/bin. On a Unix based system please follow the instructions below.

```
CHECKOUT

Make directory
 mkdir holmes
```

```
Then type
 cd holmes

Now obtain all the components that make up the compiler
  svn checkout https://subversion.cs.uu.nl/repos/staff.jur.lvm/trunk
  mv trunk lvm
  svn checkout https://subversion.cs.uu.nl/repos/staff.jur.holmes/holmes/helium
  svn checkout https://subversion.cs.uu.nl/repos/staff.jur.holmes/holmes/compare
  svn checkout https://subversion.cs.uu.nl/repos/staff.jur.Top/trunk
  mv trunk Top


COMPILATION

The standard way of compiling Helium is as follows:
cd lvm/src
./configure   # add -host i686-apple-macosx if you happen to have an Intel Mac.
cd runtime
make depend
cd ../../..
cd helium
./configure
cd src

To make the pre-processor called sherlock
make dependSherlock
make sherlock

To make the compare tool called holmes
make dependHolmes
make holmes

# make sure make is GNU make, use gmake if it does not work.
```

## 7.3 File system

To maintain a clear separation between assignments, incarnations and submissions, Holmes will work with the following data layout. As shown below, in every assignment directory there are multiple incarnation directories. An example of an incarnation name is "0809p1" (year 08/09, first period). Every incarnation

contains multiple submissions.

```
AssignmentName/
        incarnation1/
        incarnation2/
                studentgroup1/
                studentgroup2/
                        FileA.hs
                        FileB.hs
                studentgroup3/
                        Main.hs
                        Lib.hs
        incarnation3/
```

If `incarnation3` is the most recent incarnation, the pre-processor only has to work on this incarnation. The compare tool will compare a submission from `incarnation3` with all the other submissions in `incarnation3`. It also compares `incarnation3` with `incarnation2` and `incarntion1`. A comparison between `incarnation1` and `incarnation2` will be useless because it was already done in the past. Therefore we are not interested in the result of such a comparison.

## 7.4 Pre-processor

After compilation the pre-processor named *sherlock* can be found in the bin directory. The pre-processor extracts the data needed for comparison for a complete incarnation.

### 7.4.1 Config file

First a file called **holmes-conf** must be created. In this file all possible start points of a program must be declared. Every declaration must be fully qualified and on a new line. Normally this configuration file is created in the root of an incarnation. The start points defined in the configuration file are applied to all submission in this particular incarnation. It is also possible to define the start points for a specific submission. This specific configuration file overrules the configuration file for the complete incarnation and must be saved in the root of the submission.

There are multiple ways to define the start points inside a configuration file. This applies to both the incarnation configuration file and the possible submission configuration file.

- Individual specific fully qualified start points
  Defining all start points individually and fully qualified each on a separate line.
- Function wildcard
  Defining all functions from a module as a start point at once e.g. Main.* All function declared in Main are considered a start point.
- Module wildcard
  Similar to function wildcard, only applied to the module, e.g. *.main.
- Total wildcard
  With the total wildcard, *.*, all functions in any module are considered a start point.

It is possible to use different forms of start point declarations in one configuration file. Logically in some cases a wildcard overrules a specific declaration.

The more specific the start point declarations, the better the outcome of the heuristics. Start points are used to construct a call graph (see section 3.2.3) and therefore omit unused functions. When using wild cards it is possible that none or not all of the unused functions are removed and thereby influence comparison. For example, a program consisting one module Main has the call graph as specified in figure 7.1 and the set start points defined as `Main.main`. In this case only the black part of the call graph will be generated. For the set of start points defined as `Main.*`, the complete call graph (black and gray) will be generated. This mean that a possible bogus function **foo** and the whole sub graph attached to that function is now considered useful.

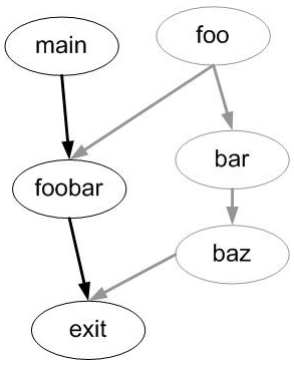


Figure 7.1: Call graph difference when using wildcard

### 7.4.2 Template code

For annotating template functions, the HolmesPragma for templates is used. There are three different ways to define a template function.

- Line based / function based
  By putting the following syntax {`-H TEMPLATE H-`} directly above the function, or on the same line as the signature if there is one, the function will me marked as template code and removed.

- Set based
  By declaring a set with function names like {`-H TEMPLATESET #function1 #function2 H-`}, where every function name has the prefix `#`, the declared functions will be marked as template code. The location of this declaration in the source is not important.

- Wildcard
  The set declaration can also be used to mark all function in a file as template code by using a star {`-H TEMPLATESET #* H-`}

It is possible to use multiple declarations in one source file. In that case the final set of template functions will be the union of all declared sets. If a function is not recognised when e.g. a function name is misspelled or the annotation is on the wrong line, it will simply be ignored and therefore not be considered a template function.

### 7.4.3 Using the pre-processor

The pre-processor is activated by calling *sherlock* with the location of the incarnation to be pre-processed. Typically the path to the Helium libraries must also be added using the `-L` flag. The library can be found in the lib folder or use the library from the Helium compiler. Calling sherlock will be like

```
sherlock -L ../lib ../../../program/incarnation1/
```

An overview of the options in the pre-processor:

- -L directory name (location of library directory)

- -C (call graph with template functions)

- -D (de chaining) (see section ...)

- -F (skip fingerprint creation) (see section 5.4)

The output of the pre-processor will be saved per submission in a subdirectory of the submission.

## 7.5 Comparing

The compare program *holmes* can also be found in the bin directory. This program takes the information created by the pre-processor and uses it for comparison. The incarnation initially given to *holmes* will be internally compared. Afterwards all the programs from the initial incarnation are compared to all the programs of previous incarnations.

To use the compare tools all the incarnations must be pre-processed. When calling *holmes* the only thing needed is the location of the initial (new) incarnation.

```
holmes ../../../program/incarnation1/
```

An overview of the options in the pre-processor:

- -F (skip fingerprint comparison)
- -C (create a CSV file of submission output)
- -T t1,t2,t3 (give thresholds for Structural, Literal en Semantic heuristics)

The output of the compare tool will be saved to disc in the root directory of the initial incarnation. By default the thresholds are set to 50 for each type of heuristic. With the `-T` flag the threshold value can be set manually for every category.

# Chapter 8

# Verification and Validation

When creating a plagiarism tool with purpose to investigate plagiarism detection in and for Haskell programs we have to be certain that the outcome of the tool is reasonably complete and correct. The implementation of the heuristics are based on assumptions we formulated based on research into similar problems. We need to both verify and validate the outcome to see if the heuristics are correct and which heuristics are relevant.

## 8.1 Verification

Verification is a quality control process. When verifying a process or product it has to evaluate according to the specifications and regulations. Simply formulated, does the process do what its supposed to do. Applying this on the Holmes heuristics: is the outcome of the heuristics correct.

There are different ways to verify a program. In our case we did not choose for a formal proof, but during development we continuously tested and monitored the input and results. In the final verification tests we measured how the heuristics act to changes in the program. We took a Helium compatible submission and applied one or multiple refactoring actions to it. Comparing all these refactored version with the original submission show us the correctness and sensitivity of the heuristics.

The following alternative versions are created by applying a single refactoring action to the original source code. The original source code is published in appendix A

| **Single refactorings** | | |
|---|---|---|
| *Name* | *Abbreviation* | *Description* |
| nameChange | nc | changed identifier names |
| translateComments | tc | translated comments from Dutch to English |
| relocation | rl | changed the order of the function declarations |
| rewrite | rw | simple transformations like `where` to `let - in` |
| trace | trc | declared a trace function similar to the Debug module and let all function call trace |
| compact | cp | move single used functions to local scope |
| unit | un | declared a unit test function that calls all functions declared in the module |

Note that not necessarily all opportunities to apply a refactoring action need to be taken. For every refactoring action in this test at least a substantial number of these opportunities are enforced.

Every transformation is influencing the defined heuristics. Therefore all the alternatives are separately implemented. Besides that, all alternatives are applied incrementally to inspect the effect when multiple refactorings are applied. The alternatives where more than one refactoring action is applied are:

| **Multiple refactorings** | |
|---|---|
| *Name* | *Implemented refactorings* |
| nc_rw | nameChange, rewrite |
| nc_rw_tc | nameChange, rewrite, translateComments |
| nc_rw_tc_cp | nameChange, rewrite, translateComments, compact |
| nc_rw_tc_cp_trc | nameChange, rewrite, translateComments, compact, trace |
| nc_rw_tc_cp_trc_un | nameChange, rewrite, translateComments, compact, trace, unit |
| nc_rw_tc_cp_trc_un_rl | nameChange, rewrite, translateComments, compact, trace, unit, relocation |

Note that the names of these multiple refactorings match with the abbreviations of the single refactorings it contains.

All alternatives are compared to the original submission. In addition to all the refactored alternatives there is also a version called `bogus`. This submission is not a refactored alternative but a completely different program to illustrate the scores to a non-plagiarised submission. The result of these tests are published in appendix B. Note that the we used rounded values in this result.

Inspecting the results show us that the scores of most of the heuristics are as expected. Due to the tokenizer and more specific the orderd token stream we can conclude that Holmes is practically insensitive for changing identifier names

(`nameChange`), relocation of definitions (relocation) and translating or deleting comments (`translateComments`). The heuristics concerning the total-degree logically fluctuate when either the in-degree or out-degree is influenced. It seems that the heuristics for the total-degree are redundant.

The values derived from the degrees may also be redundant because of the direct comparison of the degree list. We can certainly conclude that the comparisons of the minimum degrees (`idMinDiff, odMinDiff, tdMinDiff`) are completely irrelevant.

The other refactoring actions influenced the heuristics as expected. As expected, the scores are getting lower when more refactorings are applied. The refactoring actions with the highest success rate are `trace, unit` and `compact`. Although `trace` and `unit` are easily detectable by manual inspection they influence both the structural (e.g. tokenstream) and the semantic (the call graph) aspects of a submission. The same holds for `compact` because when a function is transferred from top-level scope to a local scope, both the token stream (more tokens per functions) and the call graph (less functions) are influenced. A remarkable heuristic for this particular refactoring is *fingerprinting*. It seems that *fingerprinting* is insensitive for this refactoring action although it focuses, similar to the token stream, on the structual aspects of a program. This makes *fingerprinting* a valuable heuristics. An explanation for this difference is that we sort the tokens stream per function. The creation of document fingerprinting (winnowing) handles the module (or submission) as a whole.

## 8.2 Validation

Validation is a quality assurance process. With validation you need to establish evidence that a product or process accomplishes the intended requirements. The primary objective in our case is to discover plagiarism in Haskell programs. We need to check which of the implemented heuristics can help achieve that objective.

For this test we selected two assignments from the functional program course that were Helium compatible or at least could be made Helium compatible with some minor changes. We cross compared a single iteration containing a substantial number of submissions for each of the assignments. Selecting the high score top 5 per heuristic and inspect all couples manually showed us if a particular heuristic could be considered valid. A couple in these case are the two submissions scoring high for a particular heuristic when comparing them.

The first assignment we used is called `fp-wisselkoers`. It contains 62 submission from the 2006 incarnation. The original number was higher but not all submission were used due to compatibility problems with Helium. Every submission contains two Haskell modules: *Munten.hs* and *Wisselkoers.hs*. This submission doesn't have a specific entry point therefore we specified the entry

point as `*.*` in the `holmes-conf` file.

The second assignment is called `fp-cal` and it contains 45 submission from the 2005 incarnation. The difference with the first assignment is that this one contains only one Haskell module and it has a fixed entry point *main*. Therefore the entry point specified in `holmes-conf` is set to `*.main`.

For both assignments the names of the submission are anonymised. Normally the name of a submission contains information of the student (or group). For the second assignment we picked out one submission, *Gr29* and performed the refactoring steps from the verification process on it. The submission *Copy2* contains the refactoring actions: `nameChange, translateComments, relocation` and `rewrite`. We made another submission, *Copy29_2* that contains the actions: `trace, unit` and `compact` on top of the action from *Copy29*. Both *Copy29* and *Copy29_2* are placed in a new iteration called: *Copy*.

The comparison of the submissions took several hours, at least on my old Pentium 4 2.00 GHz machine. It seems that the algorithms used to compare the degrees are the most time consuming.

### 8.2.1 Analysing results

The top 5 selection of both assignments are published in appendices D and C. When examining the top 5 for both assignments we come to the conclusion that the heuristics for token comparison and fingerprint comparison give valid results. The issue in this case is that we have to compare the scores against our own manual inspection. This doesn't mean that the rest of the result are incorrect, but there is simply not enough visual evidence for high similarity by manual inspection. When looking at the refactoring examples in section 2.3 it can get difficult to see similarity.

When inspecting *fpwisselkoers* we see that almost all heuristics score 100 when the two submissions are identical as in *Gr58 vs Gr43*. The second in line for token comparison, *Gr58 vs Gr43* is quite similar when comparing by hand but does not end up high for many other heuristics accept for fingerprinting. The same holds for *Gr47 vs Gr49* where the similarity level is less but still substantial.

When manually inspecting the top 5 of any other heuristic there is not enough similarity visible to mark it as possible plagiarism. Only the top rated comparisons for tokens and fingerprints are reliable for now. The only thing is that all submission look quite similar, probably due to the characteristics of the assignment.

The comparison results for *fpcal* are quite similar to the results for *fpwisselkoers*: only the results for token and fingerprint comparison seem to give an accurate result. The couples *Gr34 vs Gr22* and *Gr40 vs Gr5*, which are the 2 top results in both tokens and fingerprints, are showing many signs of similarity when compared manually. Even though the result for other heuristics are high

in the comparison *Gr34 vs Gr22*, we can not conclude anything from that. If we look at the results of for example `indegree 3 (ID3)`, then we see that there are other couples with a higher score. When manually inspecting these cases we can not conclude a high level of similarity.

The results for the refactored iteration *Copy* of *fpcal* (appendix C.1) shows that both submissions have a substantially high score for the token and fingerprint comparisons. For both heuristics the comparison between the two refactored cases (*Copy29* and *Copy29_2*) and the original case (*Gr29*) are in the top 3. When integrating the *Copy* scores for these two heuristics into the results for the original submission, the scores for the refactored submissions will still end up in the top 5. Again we can conclude that the heuristics for tokens and fingerprints are sufficient to detect possible plagiarism.

Not that in appendices D and C the scores that are derived from other scores, e.g. minimum indegree, are omitted. This due to publication issues (it does not fit nicely on a page) and the fact that these scores are not of any value.

# Chapter 9

# Conclusion

At the end of the project, a tool was created that can compare Helium[8] programs on different aspects. While Helium is only a subset of Haskell[6] it is still large enough to prove that we can measure software similarity in Haskell submissions.

The tool contains a substantial number of heuristics divided into three categories. On top of these heuristics the already existed method of comparing document fingerprints, as used in MOSS[14], was implemented. Our verification and validation process showed us that only the heuristics for tokens and fingerprints are reliable. Different to earlier experiments with Marble[17], comparing the token streams for Haskell programs does work with a proper normalisation process. Although the comparison between tokens and fingerprints focuses on the structural level, they are not the same. It seems that fingerprint comparison is less sensitive to changing the scope of a function from top level to local scope.

Although Literal String and Comment comparison do not conclusively find possible plagiarism, it can still be an obvious sign. If both or one of them scores high it can be wise to perform manual inspection, depending on the other results. Therefore we still believe that these two heuristics can be valuable.

It turns out that the pre-processing phase is very important. The removal of unused code by calculation of the call graph and the removal of template code reduces the number of false positives. Therefore an accurate description of the entry points of a program and template code increases the effectiveness of the tool.

The Holmes system proves that the combination of our pre-processing phase and the heuristics for tokens and fingerprints is an effective way to detected plagiarism in Haskell programs. Both Marble and MOSS do not have the ability to remove irrelevant code like Holmes does in the pre-processor. On the other hand the real value of a tool like Holmes can only be accurately judged after intensive use over a large period of time. Even so, the results thus far are certainly

promising

## 9.1 Future work

The foundation of a tool to detect plagiarism in Haskell is now created. However there are some features and changes that may improve the use and accuracy of a tool like Holmes.

At this time Holmes is built on top of Helium, this means that only Helium compatible source code will be accepted by Holmes. Helium covers only a subset of Haskell so there will be assignments that Helium will not be sufficient e.g. when IO computations are necessary. Therefore, for a wider use of the tool, it would be convenient if Holmes could accept all Haskell files that are compatible with GHC [4].

There are two more features that may improve the quality of the comparison. Because there is very little known about calculating *approximate isomorphism* between graphs, we are not able to do such a calculation on the call graph of a submission. Instead we thought that comparing the degree list would be a good alternative. We now know that the comparison of the degree list is not enough. Perhaps when there is more known about approximate isomorphism and it can be implemented, it could add some value to the current system.

Another feature we suggest is the implementation of an *Abstract Syntax Tree (AST) diff*. In the current system we transform the syntax to a token stream. The disadvantage of this method is that we lose information about the structure. If it is possible to keep the tree structure and do a diff over that structure the outcome may be more reliable than the diff over a flat token stream.

Both of these features take some implementing.

# Chapter 10

# Recommendations

To increase the effectiveness of the Holmes system it is wise that the assignments Holmes needs to check are well suited for the system. Therefore when if you happen to want to create an assignment with the purpose to use Holmes to check for possible plagiarism, we advise you to follow these recommendations.

- **Strict entry points**
  When creating an assignment make sure that you demand strict entry points. If, for example, you demand that the program has to start by executing a function called *main*, Holmes is able to do a more effective check on unused code. Because Holmes calculates the reachable functions from the given entry points, the more specific these points are, the better the result of the pre-processing.

- **Mark similar functions as template**
  If an assignment contains functions that will be exactly or almost the same in every submission, the creator should mark these function as template code. Template code will be removed in the pre-processing. When every submission contains similar function that are not marked template the structural heuristics will score unnecessarily high.

- **Don't make the assignment too strict**
  Over-specification of an assignment can lead to unintended similar outcome. If you want to achieve a certain goal and you specify all sub-functions leading to that goal as part of the assignment, then the submissions will be more similar than when you just specify the main goal. Giving a programmer some freedom on how to implement the main solution will lead to more diversity.

- **Don't make the assignment too short** When an assignment is short it is more likely that the implementation look similar. Especially in combination with over-specification this results in large numbers of similar implementation.

At this point the current tool still performs all tests when comparing two submissions. We have concluded that only the heuristics for tokens and fingerprints are reliable to test for possible plagiarism. To save time during the compare process it may be wise to remove all other forms of comparison. Especially the algorithms for comparing the degree lists are very time consuming.

# Bibliography


[1] Course functional programming. http://www.cs.uu.nl/wiki/bin/view/FP/WebHome. [cited at p. 6, 15]

[2] Course imperative programming. http://www.cs.uu.nl/docs/vakken/imp/. [cited at p. 7]

[3] The dot language. http://www.graphviz.org/doc/info/lang.html. [cited at p. 36]

[4] Glasgow haskell compiler. http://www.haskell.org/ghc/. [cited at p. 58]

[5] Graphviz. http://www.graphviz.org/. [cited at p. 36]

[6] Haskell. http://www.haskell.org/. [cited at p. 57]

[7] Haskell diff library. http://hackage.haskell.org/package/Diff. [cited at p. 41]

[8] Helium. http://www.cs.uu.nl/wiki/Helium. [cited at p. 17, 27, 45, 57]

[9] Levenshtein distance. http://www.levenshtein.net/. [cited at p. 40, 42]

[10] Lvm repository. https://subversion.cs.uu.nl/repos/staff.jur.lvm/. [cited at p. 45]

[11] Pragma's. http://www.haskell.org/ghc/docs/latest/html/users_guide/pragmas.html. [cited at p. 28]

[12] Top repository. https://subversion.cs.uu.nl/repos/staff.jur.Top. [cited at p. 45]

[13] Graduate school of natural sciences. education and examination regulations 2007-2008, 2007. http://www.science.uu.nl/naturalsciences/programmes/oer2007/GSNSOER31-8-07.pdf. [cited at p. 7]

[14] Alex Aiken. Moss. http://theory.stanford.edu/ aiken/moss/. [cited at p. 14, 20, 36, 57]

[15] Saul Schleimer; Daniel S. Wilkerson; Alex Aiken. *Winnowing: Local Algorithms for Document Fingerprinting.* Computer Science Division, UC Berkeley, 2003. [cited at p. 14, 36]

[16] Reinard Diestel. *Graph Theory.* Springer-Verlag New York, electronic edition 2000 edition, 1997, 2000. [cited at p. 19]

[17] Juriaan Hage. *Programmeerplagiaatdetectie met Marble*. Department of Information and Computing Sciences, Utrecht University, 2006. [cited at p. 8, 13, 23, 57]

[18] Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, September 2005. [cited at p. 45]

[19] Daan Leijen. *The Lazy Virtual Machine specifcation*. Institute of Information and Computing Sciences, Utrecht University, 2005. [cited at p. 45]

[20] Huiqing Li. *Refactoring Haskell Programs*. PhD thesis, University of Kent, 2006. [cited at p. 9]

[21] S. Doaitse Swierstra; Pablo R. Azero Alcocer; Joao Saraiva. *Designing and Implementing Combinator Languages*. Department of Computer Science, Utrecht University, 1999. http://people.cs.uu.nl/doaitse/Papers/1999/AFP3.pdf. [cited at p. 18]

[22] H. Li; C. Reinke; S.J. Thompson. *Tool Support for Refactoring Functional Programs*. Computing Laboratory, University of Kent, 2003. [cited at p. 9]

[23] Utrecht University. Utrecht university attribute grammar system. http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem. [cited at p. 17, 18]

# Appendices

# Appendix A

# Fql.hs

```
-- Naam: hubbie
-- Login: hhubbie
-- Studentnummer: 00000000

module Fql where

-- Definities.
---------------
type Table = (String, [String], [[String]])

top10 :: Table
top10 = ("Top10"
        , ["Nr", "Artist", "Title"]
        , [ ["1", "Mark Ronson ft. Amy Winehouse", "Valerie"]
          , ["2", "Alain Clark", "Father and Friend"]
          , ["3", "Leona Lewis", "Bleeding Love"]
          , ["4", "Kane", "Catwalk Criminal"]
          , ["5", "Colbie Caillat", "Bubbly"]
          , ["6", "Anouk", "I Don't Wanna Hurt"]
          , ["7", "Timbaland ft One Republic", "Apologize"]
          , ["8", "Lenny Kravitz", "I'll Be Waiting"]
          , ["9", "DJ Jean", "The Lauch Relaunched"]
          , ["10", "Rihanna", "Don't Stop the Music"]
          ]
        )

genre :: Table
genre = ("Genre"
        , ["Artist", "Genre"]
        , [ ["Alain Clark", "Pop"]
          , ["Anouk", "Pop"]
          , ["Anouk", "Rock"]
```

```
            , ["DJ Jean", "Dance"]
            , ["Kane", "Rock"]
            , ["Lenny Kravitz", "Rock"]
            , ["Lenny Kravitz", "Soul"]
            , ["Rihanna", "R&B"]
            ]
        )

-- Functies voor het gebruiken van 2-tuples.
-------------------------------------------
fst2 :: (a, b) -> a
fst2    (x, _) =  x
snd2 :: (a, b) -> b
snd2    (_, y) =  y

-- Functies die worden gebruikt om de breedte van de kolommen in de
    tabel te bepalen.
--
    ---------------------------------------------------------------------------


-- Geeft de totale breedte van de elke kolom in de tabel als
    resultaat.
tableWidth :: Table -> [Int]
tableWidth tabel = maxWidth (allWidth tabel)

-- Geeft de totale breedte van elke kolom van ieder record in de
    tabel en de breedte van de namen van de kolommen.
allWidth :: Table -> [[Int]]
allWidth (_, b, c) = recordWidth c ++ [singleWidth b]

-- Geeft de totale breedte van elke kolom van ieder record in de
    tabel.
recordWidth :: [[String]] -> [[Int]]
recordWidth [] = []
recordWidth (x:xs) = [singleWidth x] ++ recordWidth xs

-- Geeft de breedte van elke kolom in een record.
singleWidth :: [String] -> [Int]
singleWidth [] = []
singleWidth (x:xs) = [length x] ++ singleWidth xs

-- Geeft de maximale breedte van de kolommen in de tabel.
maxWidth :: [[Int]] -> [Int]
maxWidth [] = []
maxWidth (x:xs) = maxWidth' x xs

maxWidth' :: [Int] -> [[Int]] -> [Int]
maxWidth' [] _ = []
maxWidth' grootsten [] = grootsten
maxWidth' grootsten (x:xs) = maxWidth' (greatest grootsten x) xs
```

```
--Vergelijkt de waarden van 2 Int lists en geeft als resultaat de
    grootste getallen.
greatest :: [Int] -> [Int] -> [Int]
greatest [] [] = []
greatest [] (_:_) = [] -- Dit zou niet moeten voorkomen
greatest (_:_) [] = [] -- Idem.
greatest (x:xs) (y:ys) | x >= y = [x] ++ greatest xs ys
                       | otherwise = [y] ++ greatest xs ys

-- Functies voor printTable.
----------------------------

-- De hoofdfunctie die een tabel print.
printTable :: Table -> IO()
printTable (a, b, c) = do putStrLn (a++":")
                          putStrLn tweederegel
                          putStrLn derderegel
                          printRecords c grootsten
                       where grootsten = tableWidth (a ,b ,c)
                             tweederegel = printRecord b grootsten
                             derderegel = bigStripe grootsten

-- Print alle records in de tabel.
printRecords :: [[String]] -> [Int] -> IO()
printRecords [] _ = return()
printRecords (x:xs) grootsten = do putStrLn regel
                                   printRecords xs grootsten
                                where regel = printRecord x grootsten

-- Print een enkele record.
printRecord :: [String] -> [Int] -> String
printRecord [] [] = "|"
printRecord [] (_:_) = "|" -- Dit zou niet moeten voorkomen
printRecord (_:_) [] = "|" -- Idem.
printRecord (x:xs) (y:ys) = "|" ++ x ++ addChar (y-(length x)) ' ' ++
    printRecord xs ys

-- Geeft een ------- lijn van een bepaalde breedte als string.
bigStripe :: [Int] -> String
bigStripe [] = "-"
bigStripe (x:xs) = "-" ++ addChar x '-' ++ bigStripe xs

-- Geeft een aantal keer een char als string.
addChar :: Int -> Char -> String
addChar 0 _ = []
addChar aantal char = char : addChar (aantal-1) char

-- Functies voor count.
-----------------------
```

```
-- De hoofdfunctie die het aantal records in de tabel terug geeft.
count :: Table -> Int
count (_, _, c) = length c

-- Functies voor project.
-----------------------

-- Hoofdfunctie voor project.
project :: [String] -> Table -> Table
project lijst (a,b,c) = project' posities (a, lijst, legelijsten) (a,
    b,c)
                    where posities = getPositions b lijst
                          legelijsten = emptyLists aantal
                          aantal = count (a,b,c)

project' :: [Int] -> Table -> Table -> Table
project' [] tabel _ = tabel
project' (x:xs) (a,b,c) (d, e, f) = project' xs (a, b, voegtoe) (d,e,
    f)
                                  where voegtoe = addColumn c haalop
                                        haalop = getColumn f x

-- Geeft de positie van een string in een lijst in de vorm van [0, 1,
     2, .... ]
getPosition :: [String] -> String -> Int
getPosition [] _ = 0
getPosition (x:xs) naam | naam == x = 0
                        | otherwise = 1 + getPosition xs naam

-- Geeft de posities van meerder strings in een lijst.
getPositions :: [String] -> [String] -> [Int]
getPositions _ [] = []
getPositions lijst (x:xs) = [getPosition lijst x] ++ getPositions
    lijst xs

-- Verwijderd posities die buiten de lijst vallen.
filterPositions :: [Int] -> Int -> [Int]
filterPositions [] _ = []
filterPositions (x:xs) lengte | x < lengte = [x] ++ filterPositions
    xs lengte
                              | otherwise = filterPositions xs lengte

-- Voegt een kolom toe aan de records in een table.
addColumn :: [[String]] -> [String] -> [[String]]
addColumn [] [] = []
addColumn [] (_:_) = []
addColumn (_:_) [] = []
addColumn (x:xs) (y:ys) = [x++[y]] ++ addColumn xs ys

-- Haalt een kolom up uit de records van een table.
getColumn :: [[String]] -> Int -> [String]
```

```
getColumn [] _ = []
getColumn (x:xs) positie = [getColumn' x positie] ++ getColumn xs
    positie

getColumn' :: [String] -> Int -> String
getColumn' [] _ = []
getColumn' (x:xs) positie | positie == 0 = x
                          | otherwise = getColumn' xs (positie -1)

-- Maakt een lege lijst met lege lijsten. Nodig om kolommen toe te
    voegen aan een tabel.
emptyLists :: Int -> [[String]]
emptyLists 0 = []
emptyLists aantal = [[]] ++ emptyLists (aantal-1)

-- Functies voor select.
------------------------

-- De hoofdfunctie voor select.
select :: String -> (String -> Bool) -> Table -> Table
select naam p (a, b, c) = (a, b, resultaat)
                        where resultaat = boolRecords positie c p
                              positie = getPosition b naam

-- Voert een boolean uit op een waarde uit een record.
boolSingle :: String -> (String -> Bool) -> (Bool)
boolSingle naam p = p naam

-- Voert een boolean uit op een record.
boolRecord :: Int -> [String] -> (String -> Bool) -> (Bool)
boolRecord 0 (x:_) p = boolSingle x p
boolRecord _ [] _ = False
boolRecord positie (_:xs) p = boolRecord (positie-1) xs p

-- Voert een boolean uit op alle records.
boolRecords :: Int -> [[String]] -> (String -> Bool) -> [[String]]
boolRecords _ [] _ = []
boolRecords positie (x:xs) p | istrue = [x] ++ boolRecords positie xs
    p
                             | otherwise = boolRecords positie xs p
                            where istrue = boolRecord positie x p

-- Functies voor join.
----------------------

-- De hoofdfunctie voor join.
join :: Table -> Table -> Table
join (a,b,c) (d,e,f) | posities == (-1,-1) = error "Er zijn geen
    gemeenschappelijke kolommen"
                     | otherwise = (a ++ " " ++ d, joinRecord
                        positie2 b e, joinTableTable positie1 c
```

```
                positie2 f)
            where positie1 = fst2 posities
                  positie2 = snd2 posities
                  posities = compareTables 0 b e


-- Joined een record met een andere record.
joinRecord :: Int -> [String] -> [String] -> [String]
joinRecord positie record1 record2 = record1 ++ leaveOut positie
    record2

-- Verwijderd een kolom uit een record.
leaveOut :: Int -> [String] -> [String]
leaveOut _ [] = []
leaveOut positie (x:xs) | positie == 0 = leaveOut (positie-1) xs
                        | otherwise = [x] ++ leaveOut (positie-1) xs

-- Joined een record met meerdere records.
joinRecordTable :: Int -> [String] -> Int -> [[String]] -> [[String]]
joinRecordTable _ _ _ [] = []
joinRecordTable positie1 record positie2 (x:xs) | naam1 == naam2 = [
    joinRecord positie2 record x] ++ joinRecordTable positie1 record
    positie2 xs
                                                | otherwise =
                                                    joinRecordTable
                                                    positie1 record
                                                    positie2 xs
                                                where naam1 =
                                                    getValue positie1
                                                     record
                                                       naam2 =
                                                           getValue
                                                           positie2 x

-- Joined meerdere records met meerdere records.
joinTableTable :: Int -> [[String]] -> Int -> [[String]] -> [[String
    ]]
joinTableTable _ [] _ _ = []
joinTableTable positie1 (x:xs) positie2 tabel = joinRecordTable
    positie1 x positie2 tabel ++ joinTableTable positie1 xs positie2
    tabel

-- Leest een variabele uit een record.
getValue :: Int -> [String] -> String
getValue _ [] = []
getValue 0 (x:_) = x
getValue positie (_:xs) = getValue (positie-1) xs

-- Geeft de posities van de gemeenschappelijke kolommen uit tabellen.
compareTables :: Int -> [String] -> [String] -> (Int, Int)
```

```
compareTables _ [] _ = (-1,-1) -- als er geen gemeenschappelijke
    kolommen zijn.
compareTables start (x:xs) lijst | positie < lengte = (start, positie
    )
                                 | otherwise = compareTables (start
                                     +1) xs lijst
                                 where positie = getPosition lijst x
                                       lengte = length lijst
```

# Appendix B

# Verification table FQL

| | Tokens | indegree1 | indegree2 | indegree3 | outdegree1 | outdegree2 | outdegree3 |
|---|---|---|---|---|---|---|---|
| original VS nameChange | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS bogus | 3 | 12 | 4 | 12 | 12 | 4 | 12 |
| original VS trace | 85 | 92 | 46 | 92 | 63 | 72 | 82 |
| original VS translateComments | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS relocation | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS rewrite | 87 | 85 | 86 | 94 | 79 | 87 | 92 |
| original VS compact | 86 | 94 | 58 | 94 | 90 | 74 | 93 |
| original VS unit | 91 | 61 | 60 | 80 | 94 | 64 | 94 |
| original VS nc_rw | 87 | 85 | 86 | 94 | 79 | 87 | 92 |
| original VS nc_rw_tc | 87 | 85 | 86 | 94 | 79 | 87 | 92 |
| original VS nc_rw_tc_cp | 77 | 83 | 62 | 89 | 75 | 75 | 88 |
| original VS nc_rw_tc_cp_trc | 74 | 84 | 67 | 92 | 78 | 87 | 91 |
| original VS nc_rw_tc_cp_trc_un | 68 | 58 | 58 | 81 | 76 | 70 | 89 |
| original VS nc_rw_tc_cp_trc_un_rl | 68 | 58 | 58 | 81 | 76 | 70 | 89 |

| | totaldegree1 | totaldegree2 | totaldegree3 | idMinDiff | idMaxDiff | idAvgDiff |
|---|---|---|---|---|---|---|
| original VS nameChange | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS bogus | 12 | 1 | 12 | 100 | 11 | 43 |
| original VS trace | 62 | 69 | 86 | 100 | 56 | 82 |
| original VS translateComments | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS relocation | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS rewrite | 71 | 86 | 91 | 100 | 78 | 86 |
| original VS compact | 92 | 78 | 93 | 100 | 100 | 99 |
| original VS unit | 63 | 73 | 86 | 100 | 100 | 81 |
| original VS nc_rw | 71 | 86 | 91 | 100 | 78 | 86 |
| original VS nc_rw_tc | 71 | 86 | 91 | 100 | 78 | 86 |
| original VS nc_rw_tc_cp | 69 | 77 | 87 | 100 | 78 | 86 |
| original VS nc_rw_tc_cp_trc | 69 | 81 | 92 | 100 | 62 | 92 |
| original VS nc_rw_tc_cp_trc_un | 58 | 70 | 87 | 100 | 62 | 74 |
| original VS nc_rw_tc_cp_trc_un_rl | 58 | 70 | 87 | 100 | 62 | 74 |

| | odMinDiff | odMaxDiff | odAvgDiff | tdMinDiff | tdMaxDiff | tdAvgDiff |
|---|---|---|---|---|---|---|
| original VS nameChange | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS bogus | 100 | 33 | 43 | 100 | 22 | 43 |
| original VS trace | 100 | 90 | 82 | 100 | 50 | 82 |
| original VS translateComments | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS relocation | 100 | 100 | 100 | 100 | 100 | 100 |
| original VS rewrite | 100 | 89 | 86 | 100 | 78 | 86 |
| original VS compact | 100 | 100 | 99 | 100 | 100 | 99 |
| original VS unit | 100 | 26 | 81 | 100 | 53 | 81 |
| original VS nc_rw | 100 | 89 | 86 | 100 | 78 | 86 |
| original VS nc_rw_tc | 100 | 89 | 86 | 100 | 78 | 86 |
| original VS nc_rw_tc_cp | 100 | 89 | 86 | 100 | 78 | 86 |
| original VS nc_rw_tc_cp_trc | 100 | 100 | 92 | 100 | 55 | 92 |
| original VS nc_rw_tc_cp_trc_un | 100 | 26 | 74 | 100 | 53 | 74 |
| original VS nc_rw_tc_cp_trc_un_rl | 100 | 26 | 74 | 100 | 53 | 74 |

| | diameterDiff | parameter1 | parameter2 | parameterAvg | Strings | Comments | fingerprinting |
|---|---|---|---|---|---|---|---|
| original VS nameChange | 100 | 100 | 100 | 100 | 100 | 98 | 68 |
| original VS bogus | 40 | 0 | 0 | 0 | 0 | 0 | 0 |
| original VS trace | 100 | 97 | 97 | 100 | 98 | 99 | 68 |
| original VS translateComments | 100 | 100 | 100 | 100 | 100 | 11 | 100 |
| original VS relocation | 100 | 100 | 100 | 100 | 100 | 100 | 91 |
| original VS rewrite | 100 | 100 | 100 | 100 | 99 | 100 | 78 |
| original VS compact | 80 | 91 | 91 | 98 | 100 | 100 | 99 |
| original VS unit | 100 | 100 | 100 | 100 | 100 | 100 | 86 |
| original VS nc_rw | 100 | 100 | 100 | 100 | 99 | 98 | 53 |
| original VS nc_rw_tc | 100 | 100 | 100 | 100 | 99 | 11 | 53 |
| original VS nc_rw_tc_cp | 80 | 91 | 91 | 98 | 99 | 11 | 53 |
| original VS nc_rw_tc_cp_trc | 80 | 94 | 94 | 98 | 97 | 11 | 42 |
| original VS nc_rw_tc_cp_trc_un | 80 | 94 | 94 | 98 | 97 | 11 | 37 |
| original VS nc_rw_tc_cp_trc_un_rl | 80 | 94 | 94 | 98 | 97 | 9 | 36 |

# Appendix C

# Analyses fpcal

Top 5 for every heuristic applied on fpcal. For publication reasons not all values are printed.

Explanation of abbreviations

| | |
|---|---|
| T | Tokens |
| ID1 | in-degree algorithm 1 |
| ID2 | in-degree algorithm 2 |
| ID3 | in-degree algorithm 3 |
| OD1 | out-degree algorithm 1 |
| OD2 | out-degree algorithm 2 |
| OD3 | out-degree algorithm 3 |
| TD1 | total-degree algorithm 1 |
| TD2 | total-degree algorithm 2 |
| TD3 | total-degree algorithm 3 |
| P1 | parameters algorithm 1 |
| P2 | parameters algorithm 2 |
| S | Strings |
| C | Comments |
| F | Fingerprints |

| | T | ID1 | ID2 | ID3 | OD1 | OD2 | OD3 | TD1 | TD2 | TD3 | P1 | P2 | S | C | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TOKENS | | | | | | | | | | | | | | | |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| 2005/Gr40 VS 2005/Gr5 | 74 | 94 | 75 | 96 | 84 | 67 | 92 | 89 | 76 | 95 | 76 | 81 | 67 | 35 | 34 |
| 2005/Gr40 VS 2005/Gr33 | 71 | 77 | 50 | 85 | 73 | 32 | 85 | 77 | 49 | 88 | 67 | 91 | 55 | 29 | 30 |
| 2005/Gr40 VS 2005/Gr4 | 70 | 82 | 51 | 86 | 82 | 28 | 87 | 73 | 42 | 85 | 50 | 54 | 56 | 27 | 28 |
| 2005/Gr5 VS 2005/Gr42 | 69 | 89 | 94 | 96 | 77 | 91 | 88 | 73 | 90 | 90 | 73 | 77 | 87 | 40 | 26 |
| | | | | | | | | | | | | | | | |
| Indegree 1 | | | | | | | | | | | | | | | |
| 2005/Gr40 VS 2005/Gr5 | 74 | 94 | 75 | 96 | 84 | 67 | 92 | 89 | 76 | 95 | 76 | 81 | 67 | 35 | 34 |
| 2005/Gr40 VS 2005/Gr54 | 62 | 94 | 93 | 97 | 84 | 87 | 91 | 87 | 89 | 95 | 77 | 90 | 61 | 31 | 24 |
| 2005/Gr40 VS 2005/Gr7 | 62 | 94 | 91 | 97 | 81 | 85 | 89 | 86 | 87 | 96 | 60 | 64 | 37 | 29 | 29 |
| 2005/Gr20 VS 2005/Gr55 | 58 | 94 | 87 | 97 | 90 | 55 | 95 | 84 | 68 | 94 | 46 | 63 | 34 | 34 | 24 |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| | | | | | | | | | | | | | | | |
| Indegree 2 | | | | | | | | | | | | | | | |
| 2005/Gr54 VS 2005/Gr7 | 61 | 92 | 96 | 97 | 83 | 88 | 94 | 80 | 90 | 95 | 64 | 70 | 36 | 49 | 24 |
| 2005/Gr17 VS 2005/Gr45 | 57 | 90 | 95 | 97 | 69 | 71 | 85 | 71 | 80 | 91 | 53 | 74 | 54 | 44 | 26 |
| 2005/Gr5 VS 2005/Gr42 | 69 | 89 | 94 | 96 | 77 | 91 | 88 | 73 | 90 | 90 | 73 | 77 | 87 | 40 | 26 |
| 2005/Gr40 VS 2005/Gr54 | 62 | 94 | 93 | 97 | 84 | 87 | 91 | 87 | 89 | 95 | 77 | 90 | 61 | 31 | 24 |
| 2005/Gr53 VS 2005/Gr64 | 63 | 91 | 92 | 97 | 70 | 68 | 86 | 69 | 76 | 88 | 46 | 53 | 33 | 69 | 29 |
| | | | | | | | | | | | | | | | |
| Indegree 3 | | | | | | | | | | | | | | | |
| 2005/Gr54 VS 2005/Gr7 | 61 | 92 | 96 | 97 | 83 | 88 | 94 | 80 | 90 | 95 | 64 | 70 | 36 | 49 | 24 |
| 2005/Gr17 VS 2005/Gr45 | 57 | 90 | 95 | 97 | 69 | 71 | 85 | 71 | 80 | 91 | 53 | 74 | 54 | 44 | 26 |
| 2005/Gr40 VS 2005/Gr54 | 62 | 94 | 93 | 97 | 84 | 87 | 91 | 87 | 89 | 95 | 77 | 90 | 61 | 31 | 24 |
| 2005/Gr53 VS 2005/Gr64 | 63 | 91 | 92 | 97 | 70 | 68 | 86 | 69 | 76 | 88 | 46 | 53 | 33 | 69 | 29 |
| 2005/Gr40 VS 2005/Gr7 | 62 | 94 | 91 | 97 | 81 | 85 | 89 | 86 | 87 | 96 | 60 | 64 | 37 | 29 | 29 |
| | | | | | | | | | | | | | | | |
| Outdegree 1 | | | | | | | | | | | | | | | |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| 2005/Gr20 VS 2005/Gr55 | 58 | 94 | 87 | 97 | 90 | 55 | 95 | 84 | 68 | 94 | 46 | 63 | 34 | 34 | 24 |
| 2005/Gr59 VS 2005/Gr64 | 62 | 85 | 81 | 92 | 90 | 78 | 95 | 82 | 84 | 94 | 69 | 79 | 81 | 71 | 29 |
| 2005/Gr39 VS 2005/Gr12 | 57 | 78 | 65 | 88 | 90 | 43 | 92 | 70 | 57 | 86 | 55 | 55 | 29 | 44 | 24 |
| 2005/Gr54 VS 2005/Gr27 | 66 | 85 | 70 | 92 | 89 | 59 | 94 | 85 | 70 | 93 | 57 | 76 | 43 | 53 | 23 |
| | | | | | | | | | | | | | | | |
| Outdegree 2 | | | | | | | | | | | | | | | |
| 2005/Gr5 VS 2005/Gr42 | 69 | 89 | 94 | 96 | 77 | 91 | 88 | 73 | 90 | 90 | 73 | 77 | 87 | 40 | 26 |
| 2005/Gr46 VS 2005/Gr34 | 61 | 86 | 86 | 94 | 88 | 90 | 97 | 80 | 92 | 94 | 56 | 92 | 62 | 46 | 22 |
| 2005/Gr19 VS 2005/Gr57 | 59 | 81 | 84 | 92 | 75 | 89 | 86 | 72 | 90 | 91 | 53 | 61 | 85 | 29 | 19 |
| 2005/Gr47 VS 2005/Gr29 | 60 | 78 | 82 | 91 | 85 | 88 | 94 | 68 | 83 | 87 | 39 | 70 | 53 | 42 | 27 |
| 2005/Gr54 VS 2005/Gr7 | 61 | 92 | 96 | 97 | 83 | 88 | 94 | 80 | 90 | 95 | 64 | 70 | 36 | 49 | 24 |

| | T | ID1 | ID2 | ID3 | OD1 | OD2 | OD3 | TD1 | TD2 | TD3 | P1 | P2 | S | C | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Outdegree 3 | | | | | | | | | | | | | | | |
| 2005/Gr46 VS 2005/Gr34 | 61 | 86 | 86 | 94 | 88 | 90 | 97 | 80 | 92 | 94 | 56 | 92 | 62 | 46 | 22 |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| 2005/Gr32 VS 2005/Gr36 | 59 | 81 | 83 | 94 | 85 | 84 | 95 | 80 | 89 | 93 | 70 | 92 | 31 | 45 | 24 |
| 2005/Gr4 VS 2005/Gr35 | 68 | 86 | 84 | 93 | 86 | 80 | 95 | 81 | 86 | 93 | 61 | 72 | 61 | 32 | 28 |
| 2005/Gr32 VS 2005/Gr54 | 60 | 85 | 84 | 93 | 86 | 79 | 95 | 72 | 83 | 92 | 67 | 85 | 39 | 47 | 23 |
| | | | | | | | | | | | | | | | |
| Totaldegree 1 | | | | | | | | | | | | | | | |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| 2005/Gr40 VS 2005/Gr5 | 74 | 94 | 75 | 96 | 84 | 67 | 92 | 89 | 76 | 95 | 76 | 81 | 67 | 35 | 34 |
| 2005/Gr5 VS 2005/Gr54 | 62 | 88 | 76 | 95 | 80 | 68 | 87 | 89 | 78 | 96 | 62 | 75 | 61 | 34 | 24 |
| 2005/Gr40 VS 2005/Gr54 | 62 | 94 | 93 | 97 | 84 | 87 | 91 | 87 | 89 | 95 | 77 | 90 | 61 | 31 | 24 |
| 2005/Gr23 VS 2005/Gr19 | 59 | 89 | 48 | 90 | 87 | 45 | 90 | 87 | 59 | 90 | 53 | 59 | 33 | 36 | 27 |
| | | | | | | | | | | | | | | | |
| Totaldegree 2 | | | | | | | | | | | | | | | |
| 2005/Gr46 VS 2005/Gr34 | 61 | 86 | 86 | 94 | 88 | 90 | 97 | 80 | 92 | 94 | 56 | 92 | 62 | 46 | 22 |
| 2005/Gr47 VS 2005/Gr66 | 59 | 89 | 91 | 96 | 78 | 86 | 91 | 80 | 92 | 94 | 50 | 74 | 44 | 26 | 29 |
| 2005/Gr23 VS 2005/Gr66 | 61 | 89 | 87 | 96 | 75 | 85 | 88 | 81 | 91 | 94 | 50 | 63 | 27 | 41 | 30 |
| 2005/Gr54 VS 2005/Gr7 | 61 | 92 | 96 | 97 | 83 | 88 | 94 | 80 | 90 | 95 | 64 | 70 | 36 | 49 | 24 |
| 2005/Gr5 VS 2005/Gr42 | 69 | 89 | 94 | 96 | 77 | 91 | 88 | 73 | 90 | 90 | 73 | 77 | 87 | 40 | 26 |
| | | | | | | | | | | | | | | | |
| Totaldegree 3 | | | | | | | | | | | | | | | |
| 2005/Gr5 VS 2005/Gr36 | 63 | 89 | 85 | 95 | 80 | 81 | 89 | 87 | 89 | 96 | 70 | 80 | 52 | 43 | 27 |
| 2005/Gr54 VS 2005/Gr36 | 61 | 86 | 79 | 94 | 87 | 81 | 93 | 86 | 87 | 96 | 58 | 84 | 51 | 42 | 24 |
| 2005/Gr40 VS 2005/Gr7 | 62 | 94 | 91 | 97 | 81 | 85 | 89 | 86 | 87 | 96 | 60 | 64 | 37 | 29 | 29 |
| 2005/Gr5 VS 2005/Gr54 | 62 | 88 | 76 | 95 | 80 | 68 | 87 | 89 | 78 | 96 | 62 | 75 | 61 | 34 | 24 |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| | | | | | | | | | | | | | | | |
| Parameter 1 | | | | | | | | | | | | | | | |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| 2005/Gr12 VS 2005/Gr0 | 56 | 81 | 44 | 82 | 79 | 26 | 81 | 77 | 18 | 81 | 83 | 83 | 35 | 27 | 25 |
| 2005/Gr56 VS 2005/Gr46 | 62 | 83 | 72 | 93 | 82 | 69 | 93 | 71 | 74 | 91 | 80 | 87 | 38 | 19 | 26 |
| 2005/Gr39 VS 2005/Gr53 | 58 | 81 | 55 | 88 | 84 | 28 | 89 | 73 | 47 | 85 | 79 | 79 | 37 | 12 | 23 |
| 2005/Gr16 VS 2005/Gr64 | 62 | 83 | 53 | 90 | 83 | 52 | 91 | 76 | 61 | 89 | 78 | 94 | 55 | 39 | 29 |
| | | | | | | | | | | | | | | | |
| Parameter 2 | | | | | | | | | | | | | | | |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| 2005/Gr48 VS 2005/Gr59 | 52 | 61 | 21 | 62 | 56 | 16 | 61 | 60 | 16 | 62 | 68 | 98 | 20 | 24 | 10 |
| 2005/Gr44 VS 2005/Gr17 | 60 | 89 | 88 | 95 | 77 | 76 | 89 | 74 | 81 | 89 | 70 | 97 | 48 | 41 | 26 |
| 2005/Gr40 VS 2005/Gr17 | 63 | 84 | 47 | 86 | 81 | 47 | 82 | 81 | 60 | 86 | 77 | 96 | 80 | 35 | 27 |
| 2005/Gr17 VS 2005/Gr42 | 61 | 91 | 58 | 91 | 86 | 59 | 89 | 75 | 65 | 86 | 77 | 96 | 60 | 41 | 24 |

| | T | ID1 | ID2 | ID3 | OD1 | OD2 | OD3 | TD1 | TD2 | TD3 | P1 | P2 | S | C | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Strings | | | | | | | | | | | | | | | |
| 2005/Gr7 VS 2005/Gr0 | 58 | 84 | 56 | 85 | 83 | 35 | 84 | 66 | 46 | 78 | 25 | 25 | 100 | 20 | 24 |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| 2005/Gr56 VS 2005/Gr0 | 58 | 85 | 67 | 89 | 76 | 52 | 84 | 62 | 57 | 80 | 23 | 23 | 96 | 35 | 24 |
| 2005/Gr56 VS 2005/Gr7 | 62 | 86 | 60 | 91 | 81 | 38 | 90 | 75 | 54 | 89 | 55 | 85 | 96 | 12 | 25 |
| 2005/Gr46 VS 2005/Gr57 | 63 | 82 | 47 | 85 | 77 | 46 | 81 | 63 | 55 | 80 | 47 | 50 | 92 | 40 | 19 |
| | | | | | | | | | | | | | | | |
| Comments | | | | | | | | | | | | | | | |
| 2005/Gr37 VS 2005/Gr6 | 21 | 20 | 7 | 20 | 19 | 6 | 20 | 20 | 4 | 20 | 0 | 0 | 0 | 83 | 7 |
| 2005/Gr53 VS 2005/Gr59 | 67 | 85 | 86 | 93 | 70 | 69 | 87 | 76 | 79 | 92 | 50 | 62 | 39 | 82 | 32 |
| 2005/Gr6 VS 2005/Gr64 | 62 | 82 | 79 | 91 | 84 | 72 | 92 | 73 | 75 | 88 | 62 | 66 | 76 | 80 | 29 |
| 2005/Gr37 VS 2005/Gr59 | 26 | 19 | 7 | 19 | 20 | 7 | 20 | 20 | 3 | 20 | 0 | 0 | 0 | 78 | 8 |
| 2005/Gr6 VS 2005/Gr59 | 63 | 75 | 88 | 89 | 84 | 86 | 93 | 73 | 87 | 90 | 64 | 80 | 63 | 77 | 30 |
| | | | | | | | | | | | | | | | |
| Fingerprints | | | | | | | | | | | | | | | |
| 2005/Gr34 VS 2005/Gr22 | 94 | 93 | 76 | 96 | 94 | 57 | 97 | 92 | 69 | 96 | 92 | 100 | 98 | 40 | 42 |
| 2005/Gr40 VS 2005/Gr5 | 74 | 94 | 75 | 96 | 84 | 67 | 92 | 89 | 76 | 95 | 76 | 81 | 67 | 35 | 34 |
| 2005/Gr53 VS 2005/Gr59 | 67 | 85 | 86 | 93 | 70 | 69 | 87 | 76 | 79 | 92 | 50 | 62 | 39 | 82 | 32 |
| 2005/Gr18 VS 2005/Gr66 | 60 | 67 | 21 | 72 | 73 | 22 | 75 | 65 | 28 | 71 | 38 | 42 | 71 | 40 | 32 |
| 2005/Gr12 VS 2005/Gr66 | 54 | 71 | 24 | 72 | 72 | 23 | 73 | 65 | 30 | 70 | 23 | 23 | 80 | 39 | 32 |

## C.1 Copy iteration

| | T | ID1 | ID2 | ID3 | OD1 | OD2 | OD3 | TD1 | TD2 | TD3 | P1 | P2 | S | C | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tokens | | | | | | | | | | | | | | | |
| copies/Copy29 VS 2005/Gr29 | 93 | 81 | 54 | 90 | 84 | 39 | 91 | 70 | 50 | 87 | 61 | 100 | 74 | 1 | 55 |
| copies/Copy29 VS copies/Copy29_2 | 82 | 94 | 75 | 96 | 88 | 64 | 95 | 84 | 72 | 94 | 82 | 82 | 80 | 15 | 84 |
| copies/Copy29_2 VS 2005/Gr29 | 80 | 83 | 66 | 92 | 80 | 55 | 90 | 71 | 65 | 88 | 64 | 82 | 52 | 11 | 52 |
| copies/Copy29 VS 2005/Gr35 | 62 | 77 | 43 | 83 | 77 | 22 | 85 | 74 | 33 | 84 | 71 | 80 | 58 | 3 | 27 |
| copies/Copy29 VS 2005/Gr46 | 62 | 88 | 64 | 92 | 80 | 43 | 90 | 79 | 58 | 89 | 68 | 83 | 56 | 2 | 27 |
| | | | | | | | | | | | | | | | |
| fingerprinting | | | | | | | | | | | | | | | |
| copies/Copy29 VS copies/Copy29_2 | 82 | 94 | 75 | 96 | 88 | 64 | 95 | 84 | 72 | 94 | 82 | 82 | 80 | 15 | 84 |
| copies/Copy29 VS 2005/Gr29 | 93 | 81 | 54 | 90 | 84 | 39 | 91 | 70 | 50 | 87 | 61 | 100 | 74 | 1 | 55 |
| copies/Copy29_2 VS 2005/Gr29 | 80 | 83 | 66 | 92 | 80 | 55 | 90 | 71 | 65 | 88 | 64 | 82 | 52 | 11 | 52 |
| copies/Copy29 VS 2005/Gr40 | 60 | 86 | 78 | 92 | 83 | 66 | 90 | 81 | 74 | 93 | 63 | 89 | 92 | 2 | 28 |
| copies/Copy29 VS 2005/Gr36 | 61 | 91 | 88 | 97 | 77 | 74 | 90 | 80 | 81 | 92 | 58 | 84 | 72 | 2 | 28 |

# Appendix D

# Analyses fpwisselkoers

Top 5 for every heuristic applied on fpwisselkoers. For publication reasons not all values are printed.

Explanation of abbreviations

| | |
|---|---|
| T | Tokens |
| ID1 | in-degree algorithm 1 |
| ID2 | in-degree algorithm 2 |
| ID3 | in-degree algorithm 3 |
| OD1 | out-degree algorithm 1 |
| OD2 | out-degree algorithm 2 |
| OD3 | out-degree algorithm 3 |
| TD1 | total-degree algorithm 1 |
| TD2 | total-degree algorithm 2 |
| TD3 | total-degree algorithm 3 |
| P1 | parameters algorithm 1 |
| P2 | parameters algorithm 2 |
| S | Strings |
| C | Comments |
| F | Fingerprints |

| | T | ID1 | ID2 | ID3 | OD1 | OD2 | OD3 | TD1 | TD2 | TD3 | P1 | P2 | S | C | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TOKENS | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr31 VS Gr82 | 93 | 78 | 50 | 86 | 91 | 53 | 92 | 78 | 68 | 82 | 82 | 82 | 75 | 27 | 59 |
| Gr47 VS Gr49 | 86 | 78 | 35 | 79 | 79 | 26 | 80 | 79 | 19 | 80 | 74 | 74 | 20 | 17 | 46 |
| Gr5 VS Gr57 | 78 | 76 | 54 | 83 | 72 | 55 | 84 | 74 | 70 | 88 | 91 | 95 | 51 | 28 | 29 |
| Gr59 VS Gr0 | 75 | 81 | 81 | 91 | 83 | 88 | 93 | 80 | 89 | 91 | 88 | 90 | 0 | 0 | 31 |
| | | | | | | | | | | | | | | | |
| Indegree 1 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr71 VS Gr26 | 66 | 93 | 80 | 95 | 88 | 76 | 95 | 84 | 85 | 94 | 72 | 72 | 85 | 33 | 14 |
| Gr31 VS Gr30 | 58 | 90 | 81 | 96 | 76 | 80 | 91 | 76 | 86 | 92 | 84 | 92 | 50 | 4 | 7 |
| Gr86 VS Gr26 | 65 | 89 | 84 | 94 | 77 | 75 | 90 | 75 | 79 | 90 | 89 | 91 | 53 | 33 | 12 |
| Gr50 VS Gr18 | 63 | 89 | 92 | 97 | 79 | 84 | 94 | 68 | 89 | 88 | 76 | 85 | 52 | 0 | 21 |
| | | | | | | | | | | | | | | | |
| Indegree 2 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr50 VS Gr18 | 63 | 89 | 92 | 97 | 79 | 84 | 94 | 68 | 89 | 88 | 76 | 85 | 52 | 0 | 21 |
| Gr84 VS Gr15 | 61 | 83 | 92 | 96 | 84 | 93 | 94 | 76 | 92 | 94 | 91 | 95 | 56 | 17 | 13 |
| Gr8 VS Gr78 | 65 | 72 | 92 | 83 | 84 | 87 | 91 | 66 | 89 | 87 | 65 | 65 | 44 | 34 | 15 |
| Gr54 VS Gr30 | 70 | 84 | 90 | 92 | 74 | 86 | 90 | 68 | 87 | 90 | 81 | 84 | 75 | 0 | 17 |
| | | | | | | | | | | | | | | | |
| Indegree 3 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr50 VS Gr18 | 63 | 89 | 92 | 97 | 79 | 84 | 94 | 68 | 89 | 88 | 76 | 85 | 52 | 0 | 21 |
| Gr84 VS Gr15 | 61 | 83 | 92 | 96 | 84 | 93 | 94 | 76 | 92 | 94 | 91 | 95 | 56 | 17 | 13 |
| Gr31 VS Gr30 | 58 | 90 | 81 | 96 | 76 | 80 | 91 | 76 | 86 | 92 | 84 | 92 | 50 | 4 | 7 |
| Gr4 VS Gr12 | 63 | 86 | 87 | 95 | 76 | 86 | 92 | 74 | 86 | 94 | 91 | 95 | 68 | 14 | 16 |
| | | | | | | | | | | | | | | | |
| Outdegree 1 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr4 VS Gr15 | 56 | 81 | 85 | 92 | 92 | 88 | 97 | 81 | 90 | 94 | 79 | 89 | 65 | 12 | 13 |
| Gr31 VS Gr82 | 93 | 78 | 50 | 86 | 91 | 53 | 92 | 78 | 68 | 82 | 82 | 82 | 75 | 27 | 59 |
| Gr4 VS Gr82 | 52 | 85 | 78 | 92 | 90 | 82 | 94 | 69 | 76 | 84 | 87 | 87 | 65 | 5 | 11 |
| Gr71 VS Gr26 | 66 | 93 | 80 | 95 | 88 | 76 | 95 | 84 | 85 | 94 | 72 | 72 | 85 | 33 | 14 |
| | | | | | | | | | | | | | | | |
| Outdegree 2 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr10 VS Gr82 | 70 | 78 | 90 | 87 | 85 | 93 | 95 | 71 | 93 | 85 | 90 | 91 | 43 | 23 | 28 |
| Gr84 VS Gr15 | 61 | 83 | 92 | 96 | 84 | 93 | 94 | 76 | 92 | 94 | 91 | 95 | 56 | 17 | 13 |
| Gr71 VS Gr48 | 62 | 76 | 84 | 87 | 84 | 89 | 94 | 70 | 85 | 89 | 78 | 78 | 44 | 0 | 12 |
| Gr4 VS Gr15 | 56 | 81 | 85 | 92 | 92 | 88 | 97 | 81 | 90 | 94 | 79 | 89 | 65 | 12 | 13 |

| | T | ID1 | ID2 | ID3 | OD1 | OD2 | OD3 | TD1 | TD2 | TD3 | P1 | P2 | S | C | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Outdegree 3 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr4 VS Gr15 | 56 | 81 | 85 | 92 | 92 | 88 | 97 | 81 | 90 | 94 | 79 | 89 | 65 | 12 | 13 |
| Gr10 VS Gr82 | 70 | 78 | 90 | 87 | 85 | 93 | 95 | 71 | 93 | 85 | 90 | 91 | 43 | 23 | 28 |
| Gr4 VS Gr84 | 66 | 83 | 89 | 93 | 87 | 88 | 95 | 75 | 87 | 93 | 88 | 94 | 60 | 9 | 21 |
| Gr40 VS Gr42 | 69 | 80 | 85 | 94 | 82 | 86 | 95 | 66 | 85 | 91 | 72 | 77 | 50 | 9 | 17 |
| | | | | | | | | | | | | | | | |
| Totaldegree 1 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr50 VS Gr43 | 64 | 86 | 57 | 90 | 86 | 57 | 90 | 90 | 63 | 91 | 84 | 94 | 74 | 2 | 20 |
| Gr58 VS Gr50 | 64 | 86 | 57 | 90 | 86 | 57 | 90 | 90 | 63 | 91 | 84 | 94 | 74 | 2 | 20 |
| Gr23 VS Gr50 | 62 | 79 | 72 | 89 | 73 | 71 | 88 | 87 | 87 | 94 | 84 | 84 | 74 | 5 | 24 |
| Gr23 VS Gr43 | 62 | 84 | 73 | 89 | 73 | 45 | 82 | 87 | 63 | 91 | 84 | 84 | 68 | 25 | 20 |
| | | | | | | | | | | | | | | | |
| Totaldegree 2 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr10 VS Gr82 | 70 | 78 | 90 | 87 | 85 | 93 | 95 | 71 | 93 | 85 | 90 | 91 | 43 | 23 | 28 |
| Gr9 VS Gr51 | 71 | 76 | 80 | 90 | 68 | 85 | 89 | 85 | 92 | 97 | 76 | 82 | 78 | 16 | 21 |
| Gr84 VS Gr15 | 61 | 83 | 92 | 96 | 84 | 93 | 94 | 76 | 92 | 94 | 91 | 95 | 56 | 17 | 13 |
| Gr4 VS Gr15 | 56 | 81 | 85 | 92 | 92 | 88 | 97 | 81 | 90 | 94 | 79 | 89 | 65 | 12 | 13 |
| | | | | | | | | | | | | | | | |
| Totaldegree 3 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr9 VS Gr51 | 71 | 76 | 80 | 90 | 68 | 85 | 89 | 85 | 92 | 97 | 76 | 82 | 78 | 16 | 21 |
| Gr84 VS Gr15 | 61 | 83 | 92 | 96 | 84 | 93 | 94 | 76 | 92 | 94 | 91 | 95 | 56 | 17 | 13 |
| Gr4 VS Gr15 | 56 | 81 | 85 | 92 | 92 | 88 | 97 | 81 | 90 | 94 | 79 | 89 | 65 | 12 | 13 |
| Gr23 VS Gr50 | 62 | 79 | 72 | 89 | 73 | 71 | 88 | 87 | 87 | 94 | 84 | 84 | 74 | 5 | 24 |
| | | | | | | | | | | | | | | | |
| Parameter 1 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr18 VS Gr14 | 67 | 68 | 24 | 76 | 68 | 25 | 78 | 65 | 12 | 78 | 100 | 100 | 49 | 3 | 21 |
| Gr86 VS Gr81 | 63 | 72 | 42 | 76 | 68 | 22 | 78 | 58 | 22 | 72 | 100 | 100 | 14 | 0 | 10 |
| Gr45 VS Gr66 | 63 | 63 | 22 | 69 | 56 | 17 | 66 | 59 | 6 | 69 | 100 | 100 | 58 | 10 | 18 |
| Gr19 VS Gr82 | 63 | 83 | 82 | 94 | 80 | 87 | 92 | 70 | 85 | 90 | 97 | 99 | 56 | 3 | 12 |
| | | | | | | | | | | | | | | | |
| Parameter 2 | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr18 VS Gr14 | 67 | 68 | 24 | 76 | 68 | 25 | 78 | 65 | 12 | 78 | 100 | 100 | 49 | 3 | 21 |
| Gr86 VS Gr81 | 63 | 72 | 42 | 76 | 68 | 22 | 78 | 58 | 22 | 72 | 100 | 100 | 14 | 0 | 10 |
| Gr45 VS Gr66 | 63 | 63 | 22 | 69 | 56 | 17 | 66 | 59 | 6 | 69 | 100 | 100 | 58 | 10 | 18 |
| Gr19 VS Gr82 | 63 | 83 | 82 | 94 | 80 | 87 | 92 | 70 | 85 | 90 | 97 | 99 | 56 | 3 | 12 |

| | T | ID1 | ID2 | ID3 | OD1 | OD2 | OD3 | TD1 | TD2 | TD3 | P1 | P2 | S | C | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Strings | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr24 VS Gr46 | 60 | 72 | 60 | 83 | 78 | 36 | 85 | 69 | 35 | 80 | 71 | 71 | 100 | 17 | 18 |
| Gr4 VS Gr55 | 61 | 67 | 23 | 70 | 64 | 18 | 70 | 60 | 7 | 69 | 94 | 94 | 97 | 7 | 17 |
| Gr4 VS Gr83 | 65 | 61 | 22 | 62 | 61 | 19 | 62 | 60 | 8 | 62 | 56 | 56 | 97 | 4 | 20 |
| Gr83 VS Gr77 | 49 | 75 | 32 | 78 | 58 | 23 | 70 | 55 | 10 | 71 | 84 | 84 | 96 | 27 | 11 |
| | | | | | | | | | | | | | | | |
| Comments | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr54 VS Gr18 | 67 | 74 | 44 | 80 | 74 | 39 | 79 | 74 | 43 | 78 | 58 | 58 | 56 | 73 | 21 |
| Gr25 VS Gr54 | 73 | 78 | 58 | 91 | 87 | 77 | 93 | 77 | 75 | 92 | 86 | 92 | 63 | 67 | 23 |
| Gr69 VS Gr54 | 63 | 73 | 77 | 85 | 58 | 68 | 80 | 62 | 78 | 87 | 58 | 62 | 61 | 53 | 12 |
| Gr25 VS Gr18 | 64 | 82 | 47 | 83 | 74 | 39 | 81 | 70 | 49 | 79 | 60 | 60 | 51 | 53 | 20 |
| | | | | | | | | | | | | | | | |
| Fingerprints | | | | | | | | | | | | | | | |
| Gr58 VS Gr43 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Gr31 VS Gr82 | 93 | 78 | 50 | 86 | 91 | 53 | 92 | 78 | 68 | 82 | 82 | 82 | 75 | 27 | 59 |
| Gr47 VS Gr49 | 86 | 78 | 35 | 79 | 79 | 26 | 80 | 79 | 19 | 80 | 74 | 74 | 20 | 17 | 46 |
| Gr37 VS Gr54 | 73 | 78 | 52 | 85 | 71 | 53 | 81 | 66 | 62 | 83 | 69 | 69 | 63 | 0 | 36 |
| Gr59 VS Gr0 | 75 | 81 | 81 | 91 | 83 | 88 | 93 | 80 | 89 | 91 | 88 | 90 | 0 | 0 | 31 |

# Appendix E

# Compile and Repository instructions

```
Hello,

This is instruction file that describes how to obtain and organize the
repositories that make up the Holmes system.

First make sure a new version of helium is installed,
- either a helium system source repository from the website from after 15 jan 2010
- the helium svn repository
  (instructions @ https://subversion.cs.uu.nl/repos/staff.jur.heliumsystem/README)


CHECKOUT

Make directory
 mkdir holmes

Then type
 cd holmes

Now obtain all the components that make up the compiler
  svn checkout https://subversion.cs.uu.nl/repos/staff.jur.lvm/trunk
  mv trunk lvm
  svn checkout https://subversion.cs.uu.nl/repos/staff.jur.holmes/holmes/helium
  svn checkout https://subversion.cs.uu.nl/repos/staff.jur.holmes/holmes/compare
  svn checkout https://subversion.cs.uu.nl/repos/staff.jur.Top/trunk
  mv trunk Top


COMPILATION

The standard way of compiling Helium is as follows:
cd lvm/src
./configure   # add -host i686-apple-macosx if you happen to have an Intel Mac.
cd runtime
make depend
cd ../../..
cd helium
./configure
cd src

To make the pre-processor called sherlock
make dependSherlock
```

```
make sherlock

to make the compare tool called holmes
make dependHolmes
make holmes

# make sure make is GNU make, use gmake if it does not work.
```

# Appendix F

# Demo

## F.1 Source (Demo.hs)

```
import List

data Table = Table TableName FieldNames Records
type TableName = String
type FieldNames = [String]
type Records = [[String]]

top10 :: Table
top10 = Table "Top10"
         ["Nr", "Artist", "Title"]
         [ ["1", "Mark Ronson ft. Amy Winehouse", "Valerie"]
          , ["2", "Alain Clark", "Father and Friend"]
          , ["3", "Leona Lewis", "Bleeding Love"]
          , ["4", "Kane", "Catwalk Criminal"]
          , ["5", "Colbie Caillat", "Bubbly"]
          , ["6", "Anouk", "I Don't Wanna Hurt"]
          , ["7", "Timbaland ft One Republic", "Apologize"]
          , ["8", "Lenny Kravitz", "I'll Be Waiting"]
          , ["9", "DJ Jean", "The Lauch Relaunched"]
          , ["10", "Rihanna","Don't Stop the Music"]
          ]

printTable :: Table -> IO()
printTable = putStrLn . writeTable

writeTable :: Table -> [Char]
writeTable (Table nm fn recs) = writeName ++ writeRecord fn ++
    writeLine ++ concatMap writeRecord recs
        where
                maxwidth = columnWidth (fn : recs)
                writeName = nm ++ ": \n"
```

```
                writeRecord rec = "|" ++ fit maxwidth rec ++ "\n"
                writeLine = (replicate (foldr (+) 4 maxwidth) '-') ++
                    "\n"


fit :: [Int] -> FieldNames -> String
fit [] _ = ""
fit (x:xs) [] = (replicate x ' ') ++ "|"
fit (x:xs) (y:ys) = y ++ spaces ++ "|" ++ fit xs ys
        where
                spaces = replicate (x - (length y)) ' '

columnWidth :: Records -> [Int]
columnWidth recs = map maximum $ transpose [map length rec | rec <-
    recs]
```

## F.2 Tokens

**Demo.hs.tok**

```
X
=
X
O
X
X
(
X
X
X
X
)
=
X
O
X
X
O
X
O
X
X
X
where
X
=
X
(
X
O
X
)
X
=
X
O
S
X
X
=
S
O
X
X
X
O
S
X
=
(
X
(
X
(
O
)
I
X
)
C
)
O
S
X
[]
-
=
S
X
(
X
O
X
)
[]
=
(
X
X
C
)
O
S
X
(
X
O
X
)
(
X
O
X
)
=
X
O
X
O
S
O
X
X
X
where
X
=
X
(
X
O
(
X
X
)
)
C
X
X
=
X
X
O
X
[
X
X
X
|
X
<-
X
]
```

**Demo.hs.tks**

```
X
[]
-
=
S
X
(
X
O
X
)
[]
=
(
X
X
C
)
O
S
X
(
X
O
X
)
(
X
O
X
)
=
X
O
X
O
S
O
X
X
X
where
X
=
X
(
X
O
(
X
X
)
)
C
X
(
X
X
X
X
)
=
X
O
X
X
O
X
O
X
X
X
=
X
X
where
X
=
X
(
X
O
X
)
X
=
X
O
S
X
X
=
S
O
X
X
X
O
S
X
=
(
X
(
X
(
O
)
I
X
)
C
)
O
S
X
X
=
X
X
O
X
[
X
X
X
|
X
<-
X
]
X
=
X
O
X
```

## F.3 Meta data

**metadata.hol**

```
totaldegree:[1,1,1,1,1,1,1,1,1,1,2,2,2,3,6,7,8]
indegree:[0,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2]
outdegree:[0,0,0,0,0,0,0,0,0,0,0,0,0,3,5,5,7]
tdMin:1
tdMax:8
tdAvg:2.3529411764705883
idMin:0
idMax:2
idAvg:1.1764705882352942
odMin:0
odMax:7
odAvg:1.1764705882352942
diameter:3
parameters:[1,2,1]
```

## F.4 Comments

**Demo.hs.comm**

```
a
a
by
or
to
to
to
fit
fql
the
the
the
the
from
line
line
test
print
table
table
width
column
column
copied
record
single
single
string
string
widths
convert
correct
terminal
calculate
assignment
fieldnames
```

## F.5 Call graph

### F.5.1 dot specification

```
digraph callgraph {
"Prelude.++" [label="Prelude.++"]
"Prelude.replicate" [label="Prelude.replicate"]
"Prelude.-" [label="Prelude.-"]
"Demo.fit" [label="Demo.fit"]
"Prelude.length" [label="Prelude.length"]
"Prelude.$" [label="Prelude.$"]
"Prelude.map" [label="Prelude.map"]
"Prelude.maximum" [label="Prelude.maximum"]
"List.transpose" [label="List.transpose"]
"Prelude.+" [label="Prelude.+"]
"Demo.columnWidth" [label="Demo.columnWidth"]
"Prelude.concatMap" [label="Prelude.concatMap"]
"Prelude.foldr" [label="Prelude.foldr"]
"Prelude." [label="Prelude."]
"Prelude.putStrLn" [label="Prelude.putStrLn"]
"Demo.writeTable" [label="Demo.writeTable"]
"Demo.printTable" [label="Demo.printTable"]
"Demo.fit"->"Prelude.++"
"Demo.fit"->"Prelude.replicate"
"Demo.fit"->"Prelude.-"
"Demo.fit"->"Demo.fit"
"Demo.fit"->"Prelude.length"
"Demo.columnWidth"->"Prelude.$"
"Demo.columnWidth"->"Prelude.length"
"Demo.columnWidth"->"Prelude.map"
"Demo.columnWidth"->"Prelude.maximum"
"Demo.columnWidth"->"List.transpose"
"Demo.writeTable"->"Prelude.+"
"Demo.writeTable"->"Prelude.++"
"Demo.writeTable"->"Demo.columnWidth"
"Demo.writeTable"->"Prelude.concatMap"
"Demo.writeTable"->"Demo.fit"
"Demo.writeTable"->"Prelude.foldr"
"Demo.writeTable"->"Prelude.replicate"
"Demo.printTable"->"Prelude."
"Demo.printTable"->"Prelude.putStrLn"
"Demo.printTable"->"Demo.writeTable"
}
```

### F.5.2 visual call graph

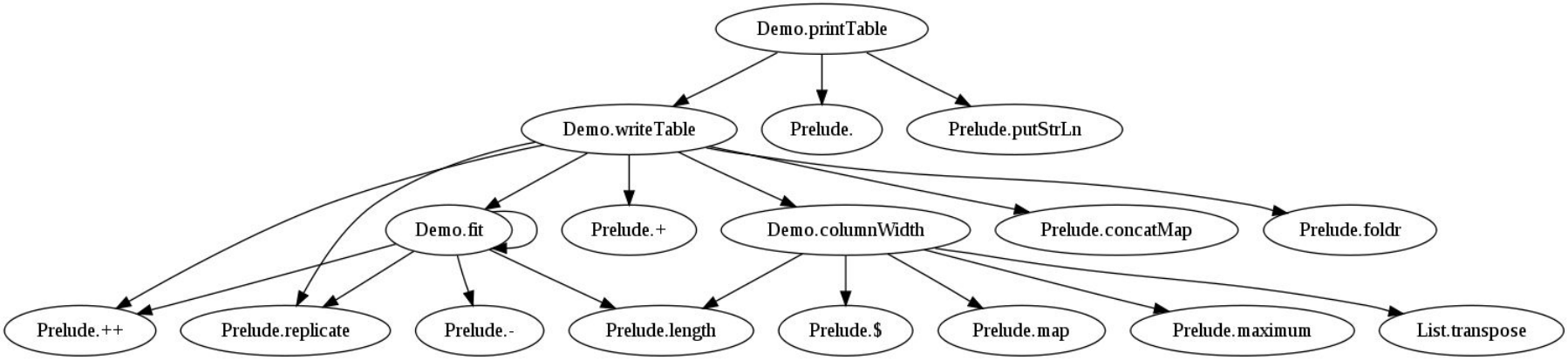


Figure F.1: visual callgraph

# List of Figures