Research paper: Plagiarism Detection for Haskell with Holmes

JURRIAAN HAGE and BRIAN VERMEER and GERBEN VERBURG
Utrecht University

Holmes is a plagiarism detection tool for Haskell programs. In this paper, we describe Holmes and show that it can detect plagiarism in a substantial corpus (2,122 Haskell submissions spread over 18 different assignments) of Haskell programs submitted by undergraduate students in a undergraduate level functional programming course over a period of ten years, and consider its sensitivity to superficial changes in the source code.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming—; K.3.2 [**Computers and Education**]: Computer and Information Science Education —*Computer science education*; D.2.3 [**Software Engineering**]: Metrics—*Complexity Measures*

General Terms: Languages, Measurement, Experimentation

Additional Key Words and Phrases: plagiarism detection tools, software analysis, Haskell, empirical validation

1. INTRODUCTION

Plagiarism is the act of copying (and sometimes superficially modifying) work of others and submitting this as one's own. At Utrecht University, if a student is found to be guilty of committing plagiarism, this can have serious consequences (from exclusion from the course for a year, and disqualification for the honours predicate, to being excluded from all courses for a year and being requested to leave the curriculum altogether). Clearly, plagiarism is regarded as an important issue, and we believe that in that case an effort should be made to actively detect whether plagiarism has taken place. Since many of our courses count their students in the hundreds, manual discovery of plagiarism can only be accidental. Moreover, an assignment may be given to students (relatively) unchanged over a period of years, which makes it possible for students to copy from code produced (and graded) in the past (and as our experiments show, they do). This implies that automation is absolutely essential to discover cases of plagiarism.

Fortunately, there are many tools out there that can help detect plagiarism. Indeed, there is an abundance of software for discovering plagiarism in essays (e.g., Ephorus[Ephorus 2012]), and for courses on programming, there are quite a few tools for discovering programming plagiarism. There are, however, not many such tools that work for Haskell programs (`www.haskell.org`). We only know of Moss [Schleimer et al. 2003] and now there is Holmes.

Constructing a plagiarism detection tool would be easy if students stuck to making exact copies of programs. However, we have observed that they add comments, change or translate them, change identifier names, or a piece of thus modified code is incorporated in a body of self-written source code. Many such changes make little or no demands on an understanding of the copied code, or the application domain. For example, comments can be easily translated from English to Dutch, or vice versa, without a real understanding of the code. As our experiments in Section 5.2 show, this is indeed a tactic often followed by students. Our experiments show that Holmes can also help unveil other forms of fraud, e.g., that in a group of two students one student is responsible for the vast majority of the work, while another comes along for the ride.

This paper discusses the features, use and limitations of Holmes, and provides some details on how Holmes works internally. For the most part, we consider how well Holmes performs. In the first half of this empirical study we apply Holmes to a large corpus of student programs submitted as part of our mandatory undergraduate Functional Programming course. The corpus contains in excess of two thousand submitted programs, from a total of 18 different assignments and were collected over a period of more than ten years. With Holmes we found 66 clear cut cases of plagiarism, and 12 cases that were less clear cut. The second half of our empirical validation is to consider how robust Holmes is against various kinds of program refactorings, by means of a sensitivity analysis. In particular, we consider what happens to the scores provided by Holmes when changes are made that require little or no understanding of the program, e.g., changing the names of identifiers, translating the comments, and reordering function definitions.

As do most such tools, Holmes detects plagiarism by means of heuristics that score either pairs of modules or pairs of submissions with some number, typically between 0 and 100. Due to their approximative nature, heuristics suffer from both false positives (flagged as plagiarism, but the similarity is due to other reasons) and false negatives (not flagged, but when examined by a teacher considered to be plagiarism). Clearly, both false positives and negatives should be avoided as much as possible. Usually, tools simply generate a long list of pairs, without really considering whether the listed pairs actually constitute plagiarism; this judgement is left to the assessor. What every tool aims to optimise is that submissions that show a high degree of similarity (note that not all similarities need be explained by acts of plagiarism) end up at the top of the list. The top-most pairs will then be subjected to manual inspection by the assessor, until the assessor has found a certain number of false positives (if we can call them that). The quality of a plagiarism tool is then the likelihood that after dismissing k (typically k is between 3 and 5) false positives, all the potentially suspect cases have already been considered.

Address: Dept. of Inf. and Comp. Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: J.Hage@uu.nl, uu@brianvermeer.nl, g.verburg@students.cs.uu.nl

DOI 10.1145/XXXXXXX.YYYYYYY
http://doi.acm.org/10.1145/XXXXXXX.YYYYYYY

The ideal situation for an empirical study is to have a ground truth, which tells us precisely which pairs of submissions are considered to be plagiarised and which are not; such a ground truth can for example be obtained by collecting expert opinions for all cases. However, given that we have over 2,000 submissions in our case study, obtaining the ground truth is an inordinate amount of work. Instead, we settle in this paper counting the number of (manually confirmed) cases we did find, and describing them in a way that hopefully convinces the reader that Holmes finds cases of plagiarism that would be hard to detect manually.

Currently, Holmes is only available from the first author to lecturers of functional programming. Soon, however, we wish to publish the tool on Hackage (the main source repository for Haskell), so that anybody can freely download, install and use it. This then includes, possibly, novice students. We are aware that this allows students to make use of Holmes to verify that any modifications they have made to other people's work decreases scores significantly. We have weighed this fact against the following considerations:

—novice programmers are typically not aware of the existence of Hackage, and if they do, then they still need to discover the existence of this tool there,

—the burden of deploying new versions of Holmes is much smaller due to our use of Hackage. Otherwise, new versions must be communicated to a group of users, and we should also keep track of those users,

—by making the tool more easily accessible, we actually hope to attain feedback from a large group of people on the cases in which Holmes does not work well.

In the end, we believe the advantages of publishing Holmes openly are larger than the disadvantages.

Paradoxically, we hope that the threat of Holmes is enough to make sure there is no plagiarism at all. Dealing with plagiarism may involve contacting the students and the examination committee, which is tedious and unproductive. If students believe that you can detect plagiarism well and easily, this will deter them from plagiarising. Everyday experience with Marble (a plagiarism detection tool for Java and C# [Hage et al. 2011]) shows that there are always a few who think they might get lucky, so that word of mouth may warn off the others.

The paper is structured as follows. We first discuss the notions of plagiarism and fraud in Section 2. In Section 3 we discuss the features of Holmes at a high-level, and Section 4 considers the heuristics it employs in more detail. Empirical data can be found in Sections 5.1 and 5.2. Related work, in particular the Moss system, is discussed in Section 6 and we conclude in 7.

2. WHAT IS PLAGIARISM?

Before we go on, we must first define what we consider to be plagiarism, and its relation to fraud. In the (master) statute of the Department of Computer and Information Science [Graduate School Of Natural Sciences 2009] of Utrecht University, fraud and plagiarism are, extremely broadly, defined as follows:

> 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.

Essentially, programming assignments are intended to assess the programming skills and the understanding of the application domain of the student. Clearly, direct copying of source code violates this definition, particularly if the copied code is central to the programming skills being assessed. We note that the construction of code is markedly different from understanding it. If the goal of programming is to verify that somebody can construct a similar program if left to his own devices, then certainly only selecting from and modifying somebody else's program does not allow the teacher to make a proper assessment. This is a grey area: at some point, the student may have changed the code so much, and have shown so much understanding of the program, the language, and the application, that the student may well have written the program from scratch. This is, in the end, up to the teacher to decide.

Keeping in mind that it is always the assessor who decides what constitutes fraud or plagiarism, the goal of a plagiarism detection tool is not to pinpoint cases of plagiarism, but to dismiss, as much as possible, cases where plagiarism is highly unlikely. In a collection of n students, there are $n(n-1)/2$ possible cases of plagiarism. In Utrecht, n may range from 60 up to 135, which makes the number of cases to consider manually uneconomical, or even infeasible. Moreover, the same assignment may be given to students unchanged over a period of years, which makes it possible for students to copy from code produced (and graded) in the past (and as our experiments show, they do).

There are a few crucial realizations to make at this point, especially because they make the life of a plagiarism detection tool much easier. The first of these facts is based on the first author's experience in talking to students who commit plagiarism:

> **Observation 1:** *A student who plagiarises typically suffers from a lack of time, a lack of programmer experience or both.*

A corollary is that students who plagiarise typically do not have the time or the knowledge to modify a submission beyond recognition while retaining its semantics. In our experience (with Java and Marble) students may translate or remove comments, translate identifier names, change indentation, or reorder method definitions. Only a few go as far as to reorder even statements in methods if they are easily seen to be reorderable.

Second, since every submission must be graded, we have that

> **Observation 2:** *A modification of a submission must be reasonable, at the very least human-readable.*

In other words, widely available code obfuscators can not help the student.

Note however that these facts together do not make plagiarism detection trivial. For example, an expert programmer may help bad programmers commit plagiarism by constructing a tool that automatically transforms a submission to something that cannot be (easily) traced back to the original, and still will pass by the grader unnoticed. In fact, as refactoring tools mature, it may well be that these tools can play exactly that unintended role. One of the goals of developing a plagiarism detection tool is to counter the power of currently available refactoring tools. In the current study, we do not pay much attention to that particular issue, although some of our heuristics are insensitive to typical refactoring steps.

Our final observation is the following:

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

A student may need to spend a long time changing a submission, because a submission must typically be changed substantially to avoid detection. For example, even if a tool is not well-suited to deal with students who reorder definitions, many definitions need to change place in order to fool a system that does nothing to deal

with reordering. Given the usual time constraints, this can hardly be done, unless supported by (refactoring) tools. One may consider a plagiarism detection tool to be succesfull if the effort spent on modifying a copy of someone else's work takes more time than writing one from scratch.

Observation 3 can be interpreted in a different way as well: if we have two very different heuristics to help us tell what is plagiarism, scoring high on any single one of these may be enough to trigger further investigation. In our experience, a manual investigation quickly tells us whether the plagiarism is there for real, or whether the similarities are explained by other reasons.

3. FEATURES AND LIMITATIONS

In this section we describe both the features and limitations of Holmes. Holmes supports detemplating, module level and submission level comparison, ignoring unreachable code and historical comparisons. As to limitations, Holmes can only compare parse correct programs, and does not compare submitted programs to online sources.

The main idea of Holmes, and many other plagiarism detection tools, is to ignore anything that can be easily changed without (essentially) changing the program. Typical examples are: comments, the order of functions and identifier names. Note however that identifier names may be irrelevant to some extent, but not entirely: since the program must be understandable (Observation 2), it is unlikely that a programmer will introduce the well-known function *map* under a different name to escape detection. And even so, doing this once or twice won't help, because of Observation 3. Therefore, it may be wise not to thoughtlessly replace all identifiers, but to retain a few of the more important and well-known ones (types like *String* and type classes like *Eq* are good examples). In Holmes, the list of identifiers to be retained can be easily modified.

Holmes supports various ways of what we call *detemplating*. Detemplating is necessary when the assignment of students is to complete a given partial solution, the template. By annotating the parts that belong to the template, we can prevent the obvious similarities between those parts from dominating the comparison. Experience with Marble has shown that if code templates are present, the similarities between submissions that arise through plagiarism are drowned out by the similarities due to the template. Since many of the assignments we hand out in our functional programming course are based on code templates, it is essential to make sure they can be dealt with. Template specifications may also be used in cases where the code is simply not very important to the assignment and is better ignored (e.g., a pretty printer in a compiler construction assignment), or the assessor expects students to provide very similar implementations (stacks and queues are a typical example).

In Holmes, there are various pragma's to specify template code: line based, set based, and with a wild card. In the former case, the text `{-H TEMPLATE H-}` right above the function definition, or on the line of its explicit type signature marks that identifier as template code. The second option is written `{-H TEMPLATESET #f1 #f2 ... #fn H-}` which marks every `fi` as a template function. The wild card template pragma, `{-H TEMPLATESET #* H-}` marks every function in the module as template code. 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.

A distinct advantage of Moss is that it can automatically disregard code found in more than k submissions (for any k). In other words, detemplating is attained for free. On the other hand, Holmes performs a simple form of reachability analysis, and this feature is not offered by Moss. Consider for example,

$$
\begin{aligned}
&\mathit{useful} = ...\\
&\mathit{spurious} = ...\\
&\mathit{cleverlyHidden} = \cdots\\
&\mathit{main} =\\
&\quad \textbf{let}\\
&\qquad f = (\mathit{spurious}, \mathit{useful})\\
&\qquad g = \mathit{cleverlyHidden}\\
&\qquad h = \mathit{useful}\\
&\quad \textbf{in } \mathit{const}\ h\ g
\end{aligned}
$$

If *main* is known to be the entry point to the program, then it can be easily verified by a simple work list algorithm that the function f is never used to compute the value of *main*. Assuming there is no other reference to *spurious* anywhere, *spurious* is dead code, which leads to Holmes ignoring the code of *spurious* in the comparison (although it will keep the local definition of f). We want to ignore *spurious* in the comparison, because a student can otherwise easily hide relatively small chunks of similar code among a large mass of seemingly innocuous but also unnecessary code.

Since our analysis is pretty simple (essentially we compute the transitive closure of the call-relation between top level definitions), spurious code can still be added. For example, g refers to the function *cleverlyHidden* (and transitively anything it might call), the result of which is then dismissed by *const*. Holmes cannot detect these cases. Implementing a more precise form of dead code analysis remains future work. We are not yet convinced of the added value of this analysis, also because a more refined analysis is significantly more complicated to implement (e.g., by means of a type and effect system [Lucassen and Gifford 1988; Holdermans and Hage 2010]), and will also be much more costly to execute. It should be noted, however, that the plagiarist has been forced to make use of *cleverlyHidden* in the body of the let explicit. Hopefully the assessor will find during grading that the call to g can actually never happen, and will wonder why that may be.

To gain anything from our reachability analysis, we provide a facility to the assessor to choose a restricted set of entry points to the program. Reachability is then considered from these entry points only. It is advisable that the assessor restricts the entry point for the assignment to one single entry point (say *main*) in one particular module (say *Main*).

In Holmes, information about entry points can currently be provided in a configuration file, and can take one of three full qualified forms: `Main.main` says that the function *main* of module *Main* is an entry point, `Main.*` specifies that all top-level functions in the module `Main` are entry points; and `*.main` specifies that all functions called `main` are entry points, wherever they may be defined.

Based on this information, a preprocessor removes all code that

—is part of the template (typically, because it may be expected that the template code is identical across students), or

—is not reachable from one of the entry points.

The output is a collection of stripped source files, for which a number of characteristics can now be computed (by a program called `holmes-prepare`). The results can then be compared among submissions and among modules by a second application `holmes-compare`.

3.1 How to use Holmes

Our experiences with Java show that many cases of plagiarism involve assignments handed in previous incarnations of the assignment. Therefore, a plagiarism detection tool must be able to compare efficiently to batches of assignments from previous years, i.e., preferably without comparing these older assignments to each other all over again (According to Alexander Aiken, Moss does not have this facility). An assignment should be structured as follows:

```
assignment/incarnation/groupid/..
```

where, in the directory `groupid`, students are free to organise their code in any way they prefer.

We may assume that the previous incarnations have been considered at an earlier time, so all we have to do now is to prepare the submissions for the latest incarnation and compare the submissions amongst themselves and to those of earlier incarnations.

Holmes outputs two files: `fileoutput.csv` and `submissionoutput.csv`, which are similarly structured. Figure 1 contains an anonymized piece of an example of the latter. The columns provide names for the five measured values (between 000 and 100, with 100 indicating very high similarity), followed by the paths to the two compared submissions. We discuss the heuristics in Section 4.

The output can be easily imported into Microsoft Excel, using the semicolons as delimiters. By considering a descending sort of, say, the fingerprints column, the cases of high similarity for that heuristic can be easily found. The submissions themselves can then easily be compared manually by generic tools such as `vimdiff`, and the `JDiff` plugin of the `jEdit` editor. A complication is that these tools focus on showing literal similarities. For example, a function of which all identifiers have only been renamed, and which has been moved from the bottom to the top of the module will not be recognizable as such by these editors. It takes some practice to find the actual similarities as measured by the tool, in order to judge them sufficiently high for taking further steps, e.g., to contact the student and ask for explanations. We stress that the numbers *only serve to flag potential cases of plagiarism*, in the hope that the ones that are flagged in fact correspond to those pairs of submissions that are most similar. It is then up to the assessor to determine that the flagged submissions are, in the opinion of the assessor, too similar for the given assignment. It is this assessment, and not the numbers generated by Holmes, that should be taken to the students, (to ask for an explanation), and, if necessary, to the examination board.

To get the most of Holmes, it will help to follow a few simple rules when constructing an assignment. We discuss these next. Most plagiarism detection tools, including Holmes, focus on structural comparisons. A human being is much better at spotting striking ad-hoc similarities. For example, both programs use the same weird background colour, contain the same weird mistake, or contain the same misspellings of a word in the comments. Since the assessor must in the end explain why he/she thinks plagiarism was commmitted, it is important that the assignment provides a certain amount of freedom to the students to come up with a particular solution. For example, if all you ask for is to implement *quicksort*, then most solutions will be very similar, and there is no way you can convince anyone that plagiarism has been committed. However, if the assignment also asks students to dress up the *quicksort* program by adding some form of underspecified user interface, then the fact that that part of the solution also shows striking similarities between two assignments provides a much more convincing proof that plagiarism has been committed.

If assignments are reused over the years, it is important to keep the previous submissions around. Students sometimes put their solutions on the web, and others can find, copy and change these to suit their needs. The majority of cases of plagiarism found with Marble [Hage et al. 2011], for example, is with assignments of previous years, and Section 5.2 shows that these cases should not discounted here either. As the number of submissions grows and grows, you will find that accidental similarities start to end up with progressively higher scores. This is a sign that the assignment needs to be replaced by a new one. Phrasing the assignment in a way that allows the student some freedom while programming, will serve to avoid such signs for longer.

Finally, to make the most of the reachability analysis, assignments should be phrased in a way that the submissions have a single point of entry. If the students are provided with template code, template annotations should be included.

3.2 Limitations

Holmes can only compare programs that are at syntactically correct. Since an assessor only needs to run Holmes on programs that are to be graded, this assumption is easy to justify in practice. This does not mean that students submit only programs that are syntactically correct, and our experiments in Section 5.2 show a small percentage of parse incorrect programs. In our comparison, we have "fixed" a number of such programs, but only to be able to include them in our study of how well Holmes performs. In an everyday educational setting, programs that are not parse correct can be disregarded because the assessor will want to reject them anyway. In practice, assistants who grade assignments may sometimes "fix" syntactically incorrect programs, if the fix happens to be trivial, e.g., an multi-line comment is not properly closed. If such may be the case, then the assessor should instruct his assistants to run Holmes again to compare the modified assignment against the others.

A second limitation is that Holmes does not compare assignments to on-line resources. As explained in Section 3.1, lecturers should ensure that the assignments they give allow for a reasonable amount of creativity. This will make it less likely that close-to-complete solutions can be found on the Web. In our own experience, students do plagiarise from the Web, but typically these are solutions by other students that have done the same course and have put their solutions on-line. We can expect that these programs have been submitted in earlier years, and so will be in our collection of submitted programs. Should it happen accidentally that a solution to an assignment can be found on-line and this was not submitted before, then we may not catch the first to submit a plagiarised submission (if he/she is the only one to do so in that incarnation), but we can still find those who follow suit later.

We believe that the only workable way to overcome this limitation is to implement a web crawler to look for Haskell source code, and to store these, and their abstractions, locally in order to compare them quickly to submitted programs. A company like Ephorus does exactly the same thing for manuscripts. Holmes does not need many changes to achieve this, although it may be necessary to spend some time improving its performance: although Holmes can run a comparison for hundreds of submissions within a reasonable amount of time, we only compare submissions of which we know that they are attempts at a solution for the same assignment. This preselection is not necessarily possible for resources found on the Web. At this time, we believe the added value of on-line comparisons are far outweighed by the necessary effort to implement and maintain it. We are not aware of any programming plagiarism detection tool that performs on-line comparisions.

```
fingerprints; (sorted) tokens; indegree1; indegree2; indegree3; sub VS sub;
015; 067; 076; 048; 079; 2007/xx-yy VS 2007/zz1;
019; 067; 052; 034; 069; 2007/xx-yy VS 2001-hugs/zz2;
026; 064; 068; 068; 079; 2007/xx-yy VS 2004/zz3;
```

Fig. 1. A snippet from by-submission output

Name	*Description*
tks	token stream comparison
in1	in-degree algorithm 1
in2	in-degree algorithm 2
in3	in-degree algorithm 3
fps	Moss style fingerprints

Fig. 2. The five implemented heuristics

```
module Token where
f y = 3
main :: Int → IO ()
main x = do
  putStrLn (f (x+x))
  return ()
```

Fig. 3. Haskell example

4. HEURISTICS

In a first study, reported upon in [Vermeer 2010], we considered a large number of heuristics, divided into three categories: literal comparisons, structural comparisons and semantic comparisons. Of these, the literal comparisons between comments, strings and the like have been discontinued altogether: they are costly to compute, and they contribute little on top of the other heuristics. Of the structural comparisons we retain only the token stream heuristic and from the semantic ones we have retained three related heuristics. The heuristics, and the abbreviations we employ in this paper are listed in Figure 2(a).

The *token stream comparison* (`tks`) is the same as the heuristic employed by Marble [Hage et al. 2011]: a module or program is interpreted as a sequence of tokens (or, more precisely, of the types of tokens). We tokenize the source code to make sure that easily transformable details that do not change the meaning of the program are removed. We do want to keep certain special symbols like the double colons, arrows, curly brackets, and parentheses. Whitespace and comments are simply removed. The literal integers, characters, floating points and strings are replaced by the characters I, C, F and S, respectively. Operators are replaced by O and most identifier names will be represented by the character X. The identifiers on an exception list of widely used identifiers like `Int, String, Bool, Maybe, Either, Show, Eq` will be retained. Currently this list contains types (with their constructors) and classes defined in the Prelude, but it is easy to alter this set of identifiers. Some well known functions, like *return*, are currently treated as any other identifier. If we find that we have too many false positives, we may be able to counter this with adding well-known identifiers to the list.

To illustrate the replacement process, the following program in Figure 3 will be mangled into `X X = I X X = do X ( X ( X O X ) ) X ()` (actually, every space is a line break). This process turns a Haskell source file into a sequence of tokens separated by newlines. These token lists will be stored in two ways: "sorted" and "unsorted". Function definition order is irrelevant in Haskell which implies that the order can be changed without affecting semantics. To counter the reordering of definitions by students to escape detection, we want to normalize this order by heuristically sorting the definitions. Because this process is heuristic, we also retain the unsorted version, just in case the sorting process inadvertently messes things up (this has been known to happen in selected cases with Marble). For the sorted output of a module, the ordering of functions is based on three aspects, in descending order of importance:

i. the arity of the definition (zero for constants)
ii. the number of tokens (essentially the size of the function)
iii. and the lexicographical ordering applied to the list of tokens that make up the function.

If two functions are the same for all three, then they are so similar that their relative ordering is not likely to be important.

Implementation of the normalizing tokenisation is straightforwardly based on modifying the pretty printer: instead of printing the contents of a token, it will print a symbol that uniquely represent for the type of token (like I for integer tokens). The sorted and unsorted forms are stored in separate files, and we construct one additional file for the submission as a whole. Since there is no natural ordering among modules, we construct only a sorted version for the submission as whole.

We now turn to the comparison of two token streams. It is important to realize that the comparison takes place both at the level of modules, and at the level of the complete program. Because top-level definitions can be easily moved from one module to another, we also need to compare programs at the submission level. Because in some assignments the complexity may reside in a particular single module, and we want to catch those cases as well, Holmes also supports module to module comparison.

The similarity of two streams of tokens is computed based on the Levenshstein distance [Levenshtein 1966] between them[1], as follows:

$$totallength = (length\ sequence1) + (length\ sequence2)$$
$$tokendiff = levenshteinDistance\ sequence1\ sequence2$$
$$similarity = 100 * (totallength - tokendiff)\ /\ totallength$$

The similarity score will be 0 if the two stream are very different, and 100 if they are identical. In our implementation we employ the Haskell `Diff` library [Clover 2008].

When we started Holmes, we expected that the structural comparison on token streams would not be sufficient to detect plagiarism well enough. Therefore, we looked at a number of comparisons that are closer to the computational structure of the submission: the call graph. Since we need the call graph of the complete submission in order to compute which top-level identifiers are reachable from the provided entry points, it is easy to generate a variety of submission metrics from the call graph. Since a call

[1]The Levenshtein distance between two strings is the minimum number of edit operations needed to transform one string into the other, where an edit operation is an insertion, deletion, or substitution of a single character.

graph does not really make sense at the level of a module, we only compute its characteristics for the submission as a whole.

A simple program is given below. In its call graph, there will be three vertices for the top level identifiers, f, g and $(++)$, and edges from f to g and from g to $(++)$.

```
f :: String
f = let
      local = g "foo"
    in
      local "bar"
g :: String → String → String
g str1 str2 = str1 ++ str2
```

Note that the function *local* is not included in the call-graph, because it is a local function; we have decided to omit these for now for various reasons: keeping these make the call graph *much* larger and the comparisons thereby very time consuming. Also, the local structure of a computation is easier to change than the top-level structure. Note that dependencies between top-level functions can be through local definitions. In other words, local definitions are not ignored during the reachability analysis.

Computing the similarity between call graphs is hard. Indeed, deciding that two graphs are isomorphic is a computationally hard problem (it has not been proven to be intractable, but the general consensus is that it is [Garey and Johnson 1979]), and approximate isomorphisms (analogous to approximate string matching) are not likely to be any easier. Therefore we decided instead to compute characteristics of graphs that can be more easily compared, and thereby approximate similarity of graphs.

The call-graph based heuristics that we have continued to use in Holmes is based on the in-degree signature of a graph G (as defined later). The reason is that that particular piece of information is rather stable for the *trace* refactoring, in which many calls to a single function *trace* are sprinkled throughout the program so that intermediate values of the program execution are written to the screen. This refactoring is hard to counter, because it induces large number of small changes throughout the program. However, it only slightly affects the in-degrees of the vertices in the call-graph. To be precise, there will be edges from many functions to the *trace* function, but that is all.

The in-degree signature is obtained from a graph by computing for each vertex $v \in G$ the number of incoming edges, and to arrange these numbers in a sorted sequence: for the call graph of the small program above, we obtain $[0,1,1]$ for f,g and $(++)$ respectively. Holmes includes three different algorithms for comparing two signatures, `in1`, `in2` and `in3`. The first of these is exactly the Levenshtein distance described above for token streams but then generalized to lists of numbers. This algorithm does not take into account the magnitude of the difference in numbers. Thus, $[1,2,3]$, $[1,2,6]$ and $[1,2,88]$ are all equally different. The second algorithm, `in2`, again uses the Levenhstein distance, but on a slightly transformed degree list: the lists consist not of the degrees themselves, but the position the degrees have in the list, in the frequency of the degree. For example: $[1,2,3]$ will be transformed to $[0,1,1,2,2,2]$, e.g., the number 3 occurs at position 2, so the position 2 will be duplicated three times. As a result, when we compare $[1,2,3]$ and $[1,2,6]$, the Levenshstein distance is computed between $[0,1,1,2,2,2]$ and $[0,1,1,2,2,2,2,2,2]$. The outcome by the second algorithm is 3 instead of 1 by the first algorithm. The third algorithm uses again a different way of measuring the difference, one that focuses on the degree to which the call graphs overlap in an approximate sense. When comparing $[1,2,4]$ and $[1,2,6]$, the sequences correspond exactly on two-thirds of the list, and between the vertices where they have different degree the size of the difference is two edges. So, for $\frac{2}{3}$ they correspond and for $\frac{1}{3}$ they are different by $\frac{2}{3}$. The similarity score for the two $100*(\frac{2}{3}+\frac{2}{3}*\frac{1}{3}) = 88.888$. This can be straightforwardly generalized to vertex lists of different sizes.

Fingerprints (`fps`) are computed by the winnowing technique implemented in Moss [Schleimer et al. 2003]. The process is driven by two parameters, the k-gram size and the window size. On advice from Alexander Aiken, we have set the k-gram size for Haskell to 25, and the window size to 5.

In [Vermeer 2010], the interested reader can find the results of applying a much larger collection of heuristics to a small set of realistic programs. Based on the outcome of that study, we selected the heuristics we have just discussed. Heuristics that did not make it are the literal comparisons for strings and comments (they can be useful, but are also very time consuming to compare), the structural comparison that considers the list of arities of functions in a particular module, and many metrics related to call graphs.

5. EXPERIMENTS

After creating a plagiarism tool with the purpose of detecting plagiarism in Haskell programs we want to establish that the outcomes of the tool are reasonably complete and correct. Therefore we need to both verify and validate the outcome to see whether the tool may be sensitive to various refactorings, and if not, whether the tool can still detect real cases of plagiarism. Sensitivity is addressed first, followed by a large experimental evaluation of Holmes.

5.1 Sensitivity analysis

Sensitivity analysis proceeds by taking a single submission, refactoring that submission in many different ways and then comparing the original program to the refactorings. Scores in Holmes are always between 0 and 100, where 100 means “highly similar” and 0 means “completely different”. For some heuristics we expect that they are insensitive to certain refactorings, so we can immediately verify with a sensitivity analysis that they behave as expected. For example, in the token-stream based heuristics we remove all information in literal strings and comments, so we expect it to be insensitive to translation of comments.

In our experience with plagiarism detection tools for Java, many tools quickly degrade in precision when refactorings are combined [Hage et al. 2011]. Therefore, we consider also various combinations of refactorings.

In our study we took a single submission of the Functional Query Language Assignment (fp-fql), course year 2007, and subjected it to the refactorings listed in Figure 4, each refactoring yielding a modified version of the submission. The combined refactorings we considered are nc_rw, nc_rw_tc, nc_rw_tc_cp, nc_rw_tc_cp_trc, nc_rw_tc_cp_trc_un, and nc_rw_tc_cp_trc_un_rl. Here, for example, nc_rw_tc combines name changes (nc), local rewrites (rw) and translation of comments (tc).

All of these versions are compared to the original submission. In the comparison we also include a randomly chosen program *bogus*. The scores obtained for bogus should be very low, since it has no relation with the assignment. In Figure 5, each row denotes a refactored submission, and each column represents one of the five heuristics discussed in Section 4. Each entry in Figure 5 contains the score obtained for the given heuristic when comparing the given refactoring with the original submission. The data is obtained from

the thesis of the third author [Vermeer 2010] by keeping only the columns of the heuristics that still remain in Holmes.

Inspection of these results show that the heuristics behave as expected. For every version, there are at least a few heuristics on which the comparison to the original scores substantially higher than for *bogus*. Furthermore, most heuristics are completely insensitive to identifier name changes, translating comments and relocating functions definitions, although fingerprinting does not do so well on translating identifier names. The reason for the former is that fingerprinting does not abstract away identifier names: it has no knowledge of the language to know what may be abstracted away and what not.

The other refactorings (tr, un and cp) are more successful at lowering the scores. Most of them change the semantics, although in for us essentially unimportant ways. For the addition of trace statements throughout the program, there are still quite a few heuristics that score well. In this case, the fingerprinting technique works less well. This is most likely due to the fact that there are many changes spread over the program. Also the scores for the token stream comparison suffer a bit. The in-degree comparisons do quite well here. This is not so surprising, because only the in-degree of one single function, *trace*, is changed. In fact, this heuristic was developed precisely with this situation in mind. It is in such a refactoring that the call graph can play a role of importance. Interestingly, during the sensitivity analysis the call graph heuristics actually do quite well, and provide stable scores even when refactoring are combined. The compacting refactoring where a top-level definition is turned into a local one, is most easily recognized by fingerprinting. When considering the combined refactorings, it shows that degradation particurly for the token stream comparison is not too bad, and some of the degree signature comparisons score really well.

The reader may be interested to know how these numbers compare with the numbers found during the plagiarism check of the incarnation from which our subject program was chosen (in this check the program was compared to a total of 420 submissions, including many from previous incarnations). We have checked the high scoring submission pairs, of which one at least should come from 2007. Among these we could find some similarity, but typically due to the fact that neither submitter had done a lot, and the submissions were therefore very small. For token stream comparisons, the highest score was 81, and the first that showed substantial differences scored 78. There are quite a few pairs scoring between 78 and 68. This means that the final three refactored entries, (nc_rw_tc_cp_trc, nc_rw_tc_cp_trc_un, nc_rw_tc_cp_trc_un_rl) do not make it near to the top in the final ranking (but fortunately, they do involve quite a bit of work). For the fingerprints, the outcomes are somewhat better. In that case, the highest scores are 50, 50 and 47, all for small submissions, while the first cases that show substantial differences, the scores start at 41 and 39. This implies that the strongly refactored cases end up quite near to the top, with the exception of the final refactoring, nc_rw_tc_cp_trc_un_rl. The latter ends up in a series of five other pairs scoring 36, and may therefore well go undetected. We note that the assignment we consider here leaves little room of freedom on the part of the programmer, which ensures that separately developed assignments may also end up close together. We believe that by providing students with more freedom to fill in certain details, the accidental scores will be lower, but the scores computed by Holmes for plagiarised cases will increase. We may therefore consider this to be a kind of worst case scenario, and find the result encouraging.

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

Fig. 4. The refactorings

original VS ...	tks	in1	in2	in3	fps
nc	100	100	100	100	68
bogus	3	12	4	12	0
trc	85	92	46	92	68
tc	100	100	100	100	100
rl	100	100	100	100	91
rw	87	85	86	94	78
compact	86	94	58	94	99
unit	91	61	60	80	86
nc_rw	87	85	86	94	53
nc_rw_tc	87	85	86	94	53
nc_rw_tc_cp	77	83	62	89	53
nc_rw_tc_cp_trc	74	84	67	92	42
nc_rw_tc_cp_trc_un	68	58	58	81	37
nc_rw_tc_cp_trc_un_rl	68	58	58	81	36

Fig. 5. The sensitivity data

total submissions	2122
different assignments	18
total incarnations	36
max repeats for an assignment	7
total students	1042
max assignment for any student	11
total number of submission to submissions comparisons	230688

Fig. 6. Summary of corpus data

5.2 Validation on real submissions

We have performed a sizable study on a corpus of 2122 submissions, submitted since 2001 as part of the mandatory undergraduate functional programming course at Utrecht University. The data is summarised in Figure 6 and Figure 7[2].

Process

In the study we only perform a by-submission comparison. Since the assignments were given out before Holmes existed, there are no template indications in the source files that we could exploit. Similarly, we could not assume more restrictive starting points than `*.*`. Note that refining starting points and exploiting templates are likely to *increase* the precision of the results.

In all, there were 2150 submissions, but a number of these were not acceptable to Holmes, resulting in 2122 making it to the com-

[2]The data on students is approximate, since at some point the identifier for students was changed from account name to student number; this happened recently, so the effect is quite small.

Assignment name	incarnations	submissions
fp-afschrift	1	65
fp-afschriftgui-ghc	1	62
fp-agenda	1	78
fp-beeldverwerking-ghc	1	59
fp-creditcardvalidation	1	93
fp-fpcal	1	68
fp-fql	6	420
fp-getallen	1	95
fp-html	1	68
fp-kalender	1	6
fp-mastermind	2	156
fp-propositielogica	7	380
fp-river	1	70
fp-rocks	1	70
fp-soccer	2	52
fp-spreadsheet	1	5
fp-turtlegraphics	4	163
fp-wiki	1	74
fp-wisselkoers	1	163
fp-wxcal	1	52

Fig. 7. Incarnations and total submissions for assignment

parison. Of these 2122 there were in fact quite a few programs that did not work immediately. This was often due to file encodings that were wrong, or small syntactic mistakes in the sources. We did verify in these cases that ghc agreed with the diagnosis. There is one large category of problematic programs that ghc did accept: haskell-src-exts does not read imported modules, but sometimes it should because they contain relevant information to disambiguate infix expressions. In those cases, we copied the infix declarations into the source files.

For the syntactic issues (yes, plenty of students submit syntactically incorrect programs), we decided to fix these as long as there were few, say one or two, and the mistakes had a simple and straightforward solution. For example, a student might forget to close a nested comment. During our investigation we discovered only one class of mistakes that was due to a small bug in haskell-src-exts having to do with indentation and nested **where** clauses; again these were easy to fix. Of the remaining 28 cases that we could not "fix", 15 turned out to be empty submissions, 11 had too many syntactic problems, one case lost too much code in the lhs2TeX translation, and one generated a strange message involving TemplateHaskell (which they did not use).

Fixing the programs went hand in hand with running `holmes-prepare` on all the assignments. When an assignment had been fully prepared, we ran `holmes-compare`, which would also compare it to earlier incarnations, making sure to consider the incarnations for an assignment in chronological order.

The outcomes were put in a file `submissionoutput.csv`, which we then imported into Microsoft Excel. We then first sorted by fingerprints score, considered the top pairs until we were satisfied that we had looked at the suspicious cases (something we decided once we had looked at a few cases that did not seem suspicious), and then did the same for the token stream column.

Cases of two members of a group submitting the same assignment in the same year were not considered (we could tell from the comments, whether this was the case), as are cases of students submitting a program similar to the one they submitted in an earlier incarnation (unless the later submission included a student not involved in the earlier incarnation, or the earlier submission itself was plagiarised).

Although we have uncovered plagiarism in our studies, we did not confront any of the students with the outcomes. Many of these programs were submitted a long time ago, and many of them either quit their studies or got a degree. It was not our intention to retrospectively accuse anyone of plagiarism. At this point we only wanted to see whether we could uncover plagiarism at all on the basis of actual submissions. Ironically, we are lucky that students did actually commit plagiarism.

Outcomes

We found a total of 63 clear cut cases of plagiarism, and eleven others that need some further investigation. We also found three clear cut cases of fraud in which a student started to collaborate with a student who had delivered a very similar program in an earlier incarnation; there is one less clear cut case of this kind. There was one really strange case in which a pair of students, say John and Paul submitted a program together, but Paul also submitted a similar but not identical program in the same year. Again, questions need to be asked here. There were three cases where the students admitted to working together (which would make it fraud, not plagiarism), and there were two additional ones that may have worked together. In total, we have identified 78 cases, which is about 3,5 percent of the total number. Of these, 27 cases involve code submitted in an earlier incarnation. The 27 cases are almost exclusively restricted to the assignments `fp-propositielogica` and `fp-fql`.

Of the cases of plagiarism only seven were identical or almost identical copies. In all other cases, the plagiarism was embedded in work of their own, or the copied code was refactored in an attempt to hide the plagiarism. The most used ways to cover up are the deletion or translation of comments, and the renaming of variables. In one case, a student took an identical copy and only added unit tests. This did affect the scores, but not enough.

If we consider the students themselves, it turns out that there are a few students that show up multiple times over the years. It is hard to tell whether they are often copied from, or whether they often copied for others (or a mix thereof), since most cases date back a few years and the students are often not around anymore. However, since quite a few students were involved multiple times, the number of "convictions" would be a lot less than the 66 – 78 we have found, because given the heavy penalty for recidivism it very rarely happens that students try again (this is based on the first author's years of experience with Java plagiarists). In all there were between 141 and 149 students involved (imprecision due to approximation in student identity, as mentioned earlier), some of these were involved a total of seven times. Note, however, that a student who puts his assignment on the web (and there is nothing illegal about that), may be plagiarised quite a few times without his/her knowing.

We want to note that although the number of cases seems quite high (at least to us), the situation might have been wholly different if Holmes had been there all along. This is one reason for having Holmes in the first place: to prevent plagiarism from occurring.

What was a surprise at first to find during our experiments is that fingerprinting works really well spotting resubmissions that were quite strongly modified. In those cases the token stream comparison worked much less well. The reason we believe is that someone changing his own work will not deliberately want to avoid detection of plagiarism and will therefore see no reason to change identifier names, or move code around a lot; they will typically be more concerned with adding new code.

A choice selection of cases of plagiarism

Clearly, discussing all these cases in detail is going to be long and tedious, so we make do with a small, typical selection. In Appendix A, we have included two pieces of code that we managed to determine to be plagiarism, to illustrate that many modifications must be made in order to fool Holmes.

An interesting case was found among the submission for `fp-wisselkoers`, in which two students showed a substantial amount of alpha-renamed code, but the give away turned out to be a function called *ontdubbel* which only they had implemented under this name, and the code was *exactly* the same, including spacing and line-breaks. It seems the students tried to cover up by having two separate modules in one submission (which was the normal case), while the other submission had combined the two modules. In a file to file comparison this case might not have been so easy to spot, because both parts are of roughly the same size. In the same incarnation, there happened to be another such case, but overall they are quite rare. `fp-wisselkoers` also had a rare case of three groups submitting similar programs. Usually, when more submissions are involved this is because two groups borrowed separately from an older assignment.

In the `fp-river` assignment, the students had to solve a puzzle problem, and if they wanted, they could generalise the puzzle. In one case, the concrete puzzle was solved independently, but for the more complicated, generalised case, one of the pair borrowed from the other. This shows that a by file comparison can certainly be helpful (although we happened to find this based on the by-submission comparison).

It does not happen often, but in some cases students even retained large parts of the comments (although the submissions themselves were certainly not identical). Not removing or translating the comments is a dead give-away for the lecturer, once the tool points out that there is reason to believe there is plagiarism involved. Note that the contents of comments are *not* used by our tool in the similarity check.

In a collection of submissions from cognitive artificial intelligence students, we found a batch of cases that showed large amounts of renamed/refactored, but also a typical function was retained as is. It is surprising that they spend all this time refactoring, but do keep pieces of unique, identical code around.

In one case, `fp-mastermind` from 2007, a pair of submitters only solved a small piece of the problem they had to solve, so little that they could not be convicted by the solutions they had. However, the give away was that the students were the only ones in their batch to have a strange way of placing their semi-colons in their **do**-statements.

In the submissions for `fp-fql` three submitters in 2003 were found plagiarising from programs submitted in 2002, and failing to pass the course in 2003, did the same in 2004.

Usually, when students modify the comments they translate them (from or to Dutch), remove them, or add them. When translating they tend to summarise the contents in another language. We have seen one case in `fp-fql`, 2001, in which the submitters translated the comments word for word.

Finally, in `fp-afschrift`, a student confessed to borrowing a piece code from a particular other student. And indeed, our tool found this particular pair of submitters to be quite similar. Surprisingly, however, the other code that they did not explicitly mention as being borrowed was much too similar as well. A somewhat related example comes from a recent assignment, `fp-river`. In this case, two students, say Sarah and Jane, admitted working together, but as it turned out there was another student Morris, who had much in common with Sarah than Sarah had with Jane.

6. RELATED WORK

For our work on Marble [Hage 2006] and a comparison between tools for plagiarism detection for Java programs, we have uncovered quite a few tools that can help detect plagiarism [Hage et al. 2011]. Tools include JPLag [Prechelt et al. 2002] [Java, C, C++, Scheme, natural language], Marble [Java, C#, PHP, Perl], Moss [Schleimer et al. 2003] [too numerous to mention], Plaggie [Ahtiainen et al. 2006] [Java], Sim [Grune and Huntjens 1989] [C, Java, Pascal, Modula-2, Lisp, Miranda and natural language texts], and Yap3 [Wise 1996] [Pascal, C and LISP]. These tools have shown a certain amount of resilience over the years. A detailed comparison is beyond the scope of this paper, but it should be noted that these all tools, with a single exception are not able to handle Haskell at all. Many tools, including Yap3 and JPlag are based on Greedy-String Tiling with an optimisation called Running Karp-Rabin Matching to make it scale [Wise 1993]. The algorithm aims to "tile" one program with pieces obtained from the other. The implementation of the algorithm in JPlag computes approximations of the optimal set of tiles heuristically. This may be why experiments show that JPlag is still quite sensitive to re-ordering of code (see [Hage et al. 2011]).

The only plagiarism detection tool suitable for Haskell that we are aware of is Moss, developed by Alexander Aiken and others [Schleimer et al. 2003]. The techniques of n-gram fingerprinting and winnowing that they use are widely applicable. Holmes implements the fingerprinting technique of Moss. As it turns out, this was a good idea, because it works very well when template code and unreachable code have been removed. As explained in Section 3, Moss can perform detemplating. It, however, cannot omit dead code from the comparison, making it easier to drown out similarities between two submissions by adding different chunks of innocuous code. Moss also cannot perform historical comparisons as discussed in the current paper. Our sensitivity analysis and other experiments [Hage et al. 2011] indicate that Moss is not so well-suited when code has been moved around and, in addition to that, variables have been renamed. Moss demands that programs be sent to the Moss server for comparison, and some assessors may not feel comfortable with such a situation. In private communication Alexander Aiken stated that submitted programs are kept for 14 days, and are then deleted. In the case of Holmes, the assessor can run the experiments locally, and is in full control.

When it comes to papers on programming plagiarism and other forms of plagiarism, it will not come as a surprise that there is quite a bit, usually published in venues that deal with education issues. Such papers can be tool based, methodological and/or empirical in nature. We have found that the tool oriented papers invariably deal with more popular languages such as Java and C, and typically aim to compete with a tool such as JPlag. In the interest of space, we shall not pursue this literature further, and instead refer to [Hage et al. 2011] for further details.

Holmes was constructed in two steps: the second author implemented a large collection of heuristics for a subset of Haskell, Helium programs [Heeren et al. 2003] to be precise. This has been documented in a master's thesis [Vermeer 2010]. Holmes was then reimplemented for full Haskell for the heuristics that we consider to contribute sufficiently much.

7. CONCLUSION AND FUTURE WORK

In this paper we have described a plagiarism detection tool Holmes for Haskell. The goal of the tool is to list cases of code similarity. The assessor can then go through the most likely cases manually and decide what to do with them. After a throrough study of possible heuristics and their effectiveness we have chosen to implement a token stream based heuristic, fingerprinting and a few call graph based heuristics into a tool that is ready to be used.

We have discussed our experiences with Holmes, and have documented a large experiment on a sizable corpus of real student submissions in our first functional programming course. The results reveal a sizable number of plagiarism attempts, including cases where the students did substantial work to prevent detection and/or did additional work themselves to perfect the submission.

As to future work, there is plenty: one possible reason for the call graph heuristics to fall short is that we compare call graphs on abstractions of those graphs, thereby increasing the possiblity of false positives. Therefore, we have done some preliminary work into a more detailed comparison of call-graphs, e.g., approximate isomorphisms and including the local call graphs, to find whether they can beat the current best heuristics. A first conclusion is that such comparisons will be very time consuming, which reduces their usefulness at the scale that we want to perform them, and we have not yet been able to verify that the comparison adds anything to the existing heuristics.

A second direction to improve on the heuristics is to generalize the token-stream comparison into a diff of the abstract syntax trees. Again, such more detailed comparisons will take quite a bit more effort to implement and maintain, and execution times may substantially increase. It is also not clear that this will lead to a substantial gain in precision. A simpler way of getting more out of the token stream comparison is to experiment with different settings (like which identifiers to retain), to see if we can make the known plagiarism cases stand out more.

In terms of usability, Holmes does not provide much. The result is a list of pairs of submissions and it is then up to the assessor to compare the submissions by hand. Earlier in this paper, we suggested tools to use to make the comparison, but what is missing with these tools is some way of putting side by side those pieces of code that (transliterated or not) were found to match by the heuristic. We want to look into this issue, because it would make the life of the assessor much easier.

We have contacted Simon Thompson to have a number of refactoring experts and novices conduct a HaRe attack on Holmes, in order to find out how easily Holmes can be fooled by using HaRe to refactor the code [Li et al. 2003].

Notwithstanding, Holmes can already be used to find the plagiarists in your Haskell classes, notwithstanding the large amount of hand refactoring performed by students. We invite lecturers to contact the first author and try it out.

REFERENCES

AHTIAINEN, A., SURAKKA, S., AND RAHIKAINEN, M. 2006. Plaggie website. http://www.cs.hut.fi/Software/Plaggie/.

CLOVER, S. 2008. Haskell diff library. `http://hackage.haskell.org/package/Diff`.

EPHORUS. Accessed January 2012. `https://www.ephorus.com/`.

GAREY, M. R. AND JOHNSON, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman.

GRADUATE SCHOOL OF NATURAL SCIENCES. 2009. Education and examination regulations 2009-2010. http://www.cs.uu.nl/people/jur/EER2009-2010.pdf.

GRUNE, D. AND HUNTJENS, M. 1989. Het detecteren van kopieën bij informatica-practica. *Informatie (in Dutch) 31,* 11, 864 – 867. http://www.cs.vu.nl/ dick/sim.html.

HAGE, J. 2006. Programmeerplagiaatdetectie met marble. Tech. Rep. UU-CS-2006-062, Department of Information and Computing Sciences, Utrecht University.

HAGE, J., RADEMAKER, P., AND VAN VUGT, N. 2011. Plagiarism detection for Java: a tool comparison. In *Proceedings of the 1st Computer Science Education Research Conference (CSERC '11)*, M. Van Eekelen, P. Sloep, and G. Van der Veer, Eds. ACM Digital Library, 33–46.

HEEREN, B., LEIJEN, D., AND VAN IJZENDOORN, A. 2003. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*. ACM Press, New York, 62 – 71.

HOLDERMANS, S. AND HAGE, J. 2010. Polyvariant flow analysis with higher-ranked polymorphic types and higher-order effect operators. In *Proceedings of the 15th ACM SIGPLAN 2010 International Conference on Functional Programming (ICFP '10)*. ACM Press, 63–74.

LEVENSHTEIN, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10,* 8, 707–710.

LI, H., REINKE, C., AND THOMPSON, S. 2003. Tool support for refactoring functional programs. In *ACM SIGPLAN 2003 Haskell Workshop*, J. Jeuring, Ed. ACM, 27–38.

LUCASSEN, J. M. AND GIFFORD, D. K. 1988. Polymorphic effect systems. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 47–57.

PRECHELT, L., MALPOHL, G., AND PHILIPPSEN, M. 2002. Finding plagiarisms among a set of programs with JPlag. *J. of Universal Comp. Sci. 8,* 11, 1016 – 1038.

SCHLEIMER, S., WILKERSON, D. S., AND AIKEN, A. 2003. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. ACM, 76 – 85.

VERMEER, B. 2010. Holmes, hunting for Haskell frauds. Unpublished manuscript, `http://www.cs.uu.nl/people/jur/brianvermeer-msc.pdf`.

WISE, M. 1993. String similarity via greedy string tiling and running Karp-Rabin matching. Tech. rep., Dept. of CS, University of Sydney. December. http://vernix.org/marcel/share/RKRGST.ps.

WISE, M. J. 1996. Yap3: improved detection of similarities in computer program and other texts. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, 1996, Philadelphia, Pennsylvania, USA, February 15-17, 1996*, J. Impagliazzo, E. S. Adams, and K. J. Klee, Eds. ACM, 130–134.

APPENDIX

A. AN EXAMPLE OF PLAGIARISED CODE

In Figures 8 and 9 we display two pieces of code that are considered to be very similar by our tool. Note that the students have in fact

```
join :: Table → Table → Table
join t u = neemRij ((length t) − 1) (watIsZelfdeKolom ((length (head t)) − 1) t u) (watIsZelfdeKolom ((length (head u)) − 1) u t) t u
  -- bepaalt hoeveelste kolom de gemeenschappelijke kolom is
watIsZelfdeKolom :: Int → Table → Table → Int
watIsZelfdeKolom kolomTeller t u
  | kolomTeller ≡ −1                                  = −1
  | elemBy (eqString) ((head t) !! kolomTeller) (head u) = kolomTeller
  | otherwise                                         = watIsZelfdeKolom (kolomTeller − 1) t u
  -- vergelijkt per rij van tabel t, de tabel u
  -- kolom k van tabel t en kolom l van tabel u zijn dezelfde.
neemRij :: Int → Int → Int → Table → Table → Table
neemRij rij k l t u
  | rij ≡ −1 = [ ]
  | otherwise = (neemRij (rij − 1) k l t u) ++ (vergelijk rij ((length u) − 1) l ((t !! rij) !! k) t u)
  -- vergelijkt de rij van tabel t met alle rijen van tabel u
  -- kolomNaam is de gemeenschappelijke kolomnaam
  -- naam is de de string die in de gemeenschappelijke kolom van tabel t en in de rij die aangegeven is zit.
vergelijk :: Int → Int → Int → String → Table → Table → Table
vergelijk rij rijVanU kolomNaam naam t u
  | rijVanU ≡ −1 = [ ]
  | eqString naam ((u !! rijVanU) !! kolomNaam) = (vergelijk rij (rijVanU − 1) kolomNaam naam t u) ++
                    [(t !! rij) ++ (zetInTabel ((length (u !! rijVanU)) − 1) kolomNaam (u !! rijVanU))]
  | otherwise          = vergelijk rij (rijVanU − 1) kolomNaam naam t u
  -- voegt per rij alle kolommen van tabel u aan de nieuwe tabel toe, behalve de gemeenschappelijke kolom van tabel u
zetInTabel :: Int → Int → [String] → [String]
zetInTabel teller gemeenschappelijkeKolom lijstVanRijVanU
  | teller ≡ −1                                          = [ ]
  | teller ⩾ 0 ∧ teller ≡ gemeenschappelijkeKolom = zetInTabel (teller − 1) gemeenschappelijkeKolom lijstVanRijVanU
  | otherwise                                            = (zetInTabel (teller − 1) gemeenschappelijkeKolom lijstVanRijVanU) ++ [lijstVanRijVanU !! teller]
```

Fig. 8. A fragment of student submitted code, `fp-fql`, 2002

tried to cover up the plagiarism, but that some parts give them away. For reasons of anonymity and space some of the code has been removed.

Note that many of the formal parameters in the bottom fragment now have meaningless names, like *i* and *k*. And because of that a function like *sym* looks very little like its counterpart *zetInTabel*. Some pieces of comment have been transcribed, but only locally and not always completely. Moreover, the students have also reordered the definitions, but the code itself was only subject to alpha-renaming. Changing the code itself, typically takes some thought, and not many students prefer to take that route. Which functions are the translation of which other function can be easily determined in this case by looking at the explicit type signatures.

```
   -- geeft de kolom van tabel a die in tabel b dezelfde heading heeft
common :: Int → Table → Table → Int
common i a b | i ≡ −1 = −1
   | elemBy (eqString) ((head a) !! i) (head b) = i
   | otherwise          = common (i − 1) a b
   -- voegt per rij alle kolommen van tabel b aan de nieuwe tabel toe, behalve de gemeenschappelijke kolom k van tabel b
sym :: Int → Int → [String] → [String]
sym j k s | j ≡ −1 = [ ]
   | j ⩾ 0 ∧ j ≡ k = sym (j − 1) k s
   | otherwise     = (sym (j − 1) k s) ++ [s !! j]
   -- vergelijkt de i-de rij van tabel a met alle rijen van tabel b
   -- j-de rij van tabel b
   -- kolom k is de gemeenschappelijke heading
   -- String s is de i-de String van de gemeenschappelijke kolom van tabel a
get :: Int → Int → Int → String → Table → Table → [[String]]
get i j k s a b | j ≡ −1 = [ ]
   | eqString s ((b !! j) !! k) = (get i (j − 1) k s a b) ++ [(a !! i) ++ (sym ((length (b !! j)) − 1) k (b !! j))]
   | otherwise          = get i (j − 1) k s a b
   -- gaat alle combinaties langs door per rij van tabel a met de rijen van tabel b vergelijken
   -- i-de rij van tabel a
   -- kolom j van tabel a en kolom k van tabel b hebben dezelfde heading
joining :: Int → Int → Int → Table → Table → Table
joining i j k a b | i ≡ −1 = [ ]
   | otherwise          = (joining (i − 1) j k a b) ++ (get i ((length b) − 1) k ((a !! i) !! j) a b)
join :: Table → Table → Table
join a b = joining ((length a) − 1) (common ((length (head a)) − 1) a b) (common ((length (head b)) − 1) b a) a b
```

Fig. 9. A piece of code similar to that of Figure 8 taken from the same incarnation