Holmes for Haskell

Jurriaan Hage
J.Hage@uu.nl

Brian Vermeer
uu@brianvermeer.com

Gerben Verburg
g.verburg@students.cs.uu.nl

Department of Information and Computing Sciences, Universiteit Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands

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 functional programming course over a period of ten years, and consider its sensitivity to superficial changes in the source code.

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, disqualification for the honours predicate, to being excluded from all courses for a year and being requested to leave the curriculum altogether). If one makes such a big issue of plagiarism, it would be strange if no effort were ever made to find out whether it has taken place. Since many of our courses count their students in the hundreds, manual discovery of plagiarism can only be accidental, particularly if one also needs to compare against submissions of earlier years. The number of comparisons to be made in a group of students grows quadratic with the size of the group. 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 obtain reasonable results.

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), 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. We only know of Moss [11] and now there is Holmes.

Note that students do not stick to making exact copies of programs: comments are added, changed or translated, identifier names are changed, the copied code is incorporated in a body of self-written source code. Some of these 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. And as our experiments in Section 5 show, this is indeed a tactic often followed by students. Our experiments show that Holmes can also help unveil other forms of fraud, for example the fact 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 both the features and use of Holmes and how well Holmes does on 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. In most cases tools simply generate a long list of pairs, knowing that most of these are not plagiarism at all. 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 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.

Paradoxically, the greatest advantage of a tool like Holmes, is to make sure there is no plagiarism at all. Dealing with plagiarism may involve contacting the students and the exam 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# [6]) shows that there are always a few who think they might get lucky, but at least they will be able to inform others that it actually works.

For obvious reasons— students should not be able to verify that their refactored copied code escapes detection —, Holmes itself will only be available from the first author to lecturers of functional programming. Directions for compilation and use are included.

The paper is structured as follows. In Section 2 we discuss the features of Holmes at a high-level, and Section 3 considers the heuristics it employs in more detail. Empirical data can be found in Sections 4 and 5. Related work is discussed in Section 6 and we conclude in 7.

2 The features of Holmes

In this section we consider some of the features of Holmes in detail: the facility for detemplating, module versus submission level comparison, ignoring unreachable code and historical comparisons. We conclude with explaining how Holmes is used, and implicitly how we employed Holmes during the experiments discussed later.

Holmes supports various ways of *detemplating*. In some cases, the assignment of students is to complete a given partial solution. By annotating parts of such a solution, we can prevent these parts from dominating the comparison, thereby drowning out the essential differences between the submissions. In the presence of much template code, all assignments will become alike. Since the assignments we consider in our comparison do not have templates, we do not discuss this feature in more detail, but see [12]. A distinct advantage of Moss is that it can automatically disregard code found in more than *k* submissions (for any *k*). In other words, they get detemplating for free.

In contrast to Moss, Holmes performs a simple form of reachability analysis. Consider for example,

```
useful = .....
spurious = ....
cleverlyHidden = ···
main =
  let
    f = (spurious, useful)
    g = cleverlyHidden
    h = useful
  in const h g
```

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, if we can convince ourselves of the added value of such an analysis. 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 is currently 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`, as described in Section 3. The results can then be compared among submissions and among modules by a second application `holmes-compare`.

2.1 How to use Holmes

Holmes comes as a Cabal package that constructs both `holmes-prepare` and `holmes-compare`, and a script to clean up after use. Holmes depends on the presence of `Ghc`, and two Cabal packages

```
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;
```

Figure 1: A snippet from by-submission output

`haskell-src-exts`, and `Diff`.

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 (Moss does not seem to have this facility).

We achieve this by demanding that assignments, at top level, are structured as follows:

```
fp-fql/
  2001-hugs/
    zz2/
      Main.hs
      sub/Screen.hs
    zz5/
      ...
  2007/
    holmes-conf
    xx-yy/
      ...
      ...
    ...
```

Here, `fp-fql` refers to a particular assignment, `2001-hugs` and `2007` to two different incarnations of the assignment, and `zz2`, `xx-yy` denote separate submissions by students. A submission consists of all the `.hs` contained therein, e.g., `Main.hs` and `Screen.hs` for `zz2`.

We may assume that the previous incarnations have been considered at an earlier time, so only the submissions from the latest incarnation, 2007, need to be prepared, and `holmes-compare` will compare the submissions from 2007 amongst themselves and to those of earlier incarnations:

```
holmes-prepare 2007/
```

and then run

```
holmes-compare 2007/
```

Note the presence of a file called `holmes-conf` in `fp-fql/2007`. This is necessary to provide Holmes with information on the entry points to the assignments. The call to `holmes-compare` is from `fp-fql`, because then Holmes can prevent comparing a new submission against itself.

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 3.

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

Figure 2: The five implemented heuristics

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.

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, contain the same misspellings of a word in the comments, or mention the same student name responsible for its construction. 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 [6], for example, is with assignments of previous years, and Section 5 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, phrase your assignments in a way that the submissions have a single point of entry. If you happen to provide the students with a template, make sure to annotate it before you hand it out.

3 Heuristics

In a first study, reported upon in [12], 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 shall employ in this paper are listed in Figure 2.

The *token stream comparison* is the same as the heuristic employed by Marble [6]: 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

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

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 screws 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 [8] between them[1], as follows:

$$\begin{aligned} \mathit{totallength} &= (\mathit{length}\ \mathit{sequence1}) + (\mathit{length}\ \mathit{sequence2}) \\ \mathit{tokendiff} &= \mathit{levenshteinDistance}\ \mathit{sequence1}\ \mathit{sequence2} \\ \mathit{similarity} &= 100 * (\mathit{totallength} - \mathit{tokendiff})\ /\ \mathit{totallength} \end{aligned}$$

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 [2].

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 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 however that the 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 [3]), 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. The reason is that that particular piece of information is rather stable for the *trc* refactoring. This refactoring is generally 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

[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.

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 are computed by the winnowing technique implemented in Moss [11]. 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 [12], the interested reader can find the results of applying a much larger collection of heuristics to small set of real programs. Based on the outcome of that study, and the outcome of a sensitivity analysis also reported upon, we selected the heuristics we have just discussed. The heuristics that did not make it to second version of Holmes 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, such as the diameter of the call graph, the total degree and out degree signatures of the call graph.

4 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 an earlier study of plagiarism detection tools for Java, we found that many tools quickly degrade in precision when refactorings are combined [6]. 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 Figure 3, each refactoring yielding a modified version of the submission. The combined refactorings, abbreviated following the top table of Figure 4, were 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, local rewrites and translation of comments.

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. The results are given in Figure 4; the column headings are taken from

Figure 2. The data is obtained from the thesis of the third author [12] by keeping only the colums 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 conjecture that by providing students with more freedom to fill in certain details, the accidental scores will be lower, and the scores computed by Holmes for plagiarised cases will increase. We therefore consider this to be a "worst case scenario", and find the results encouraging.

5 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. Characteristics of

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

Figure 3: The applied 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

Figure 4: The sensitivity data

the data set are as follows: the total number of submissisions is 2122 spread over 36 incarnations based on 18 different assignments. Assignments are used up to seven times. Approximately 1042 students were involved[2]. Our empirical study included a total number of 230.688 submission to submission comparisons.

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

[2]This number is approximate, because 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	#times	#submissions	Assignment	#times	#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

Figure 5: Summary of corpus data

in 2122 making it to the comparison. 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[3].

For the syntactic issues (quite a few 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[4]. 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 at least two cases that did not seem suspicious), and then did the same for the token stream column. We did not consider the outcomes of the other three heuristics, since the numbers did not seem very indicative.

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).

[3]From Holmes version 1.0 onwards, this issue has been resolved, because as we found out later, haskell-src-exts can ignore fixity resolution.

[4]As of haskel-src-exts version 1.13.x (which Holmes employs), this issue and the issue of conditionals in a monadic `do` have been resolved.

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 in the submitted code (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 (see Appendix A for an example). In one case, a student made an exact 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.

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 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 pair use the solution of 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 assessor, 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 more in common with Sarah than Sarah had with Jane.

6 Related work

For our work on Marble [5] and a comparison between tools for plagiarism detection for Java programs, we have uncovered quite a few tools that can help detect plagiarism [6]. Tools include JPLag [10] [Java, C, C++, Scheme, natural language], Marble [Java, C#, PHP, Perl], Moss [11] [too numerous to mention], Plaggie [1] [Java], Sim [4] [C, Java, Pascal, Modula-2, Lisp, Miranda and natural language texts], and Yap3 [13] [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.

The exception is Moss, developed by Alexander Aiken and others [11]. The technique of winnowing that they use is widely applicable. Holmes implements the fingerprinting technique of Moss. As it turns out, this was a good idea, because the winnowing technique works really well when template code and unreachable code have been removed. As explained in Section 2, Moss can perform detemplating. It, however, cannot omit dead code from the comparison and cannot perform historical comparisons as discussed in this paper. Our sensitivity analysis and other experiments [6] indicate that Moss is not so well-suited when code has moved around and variables are renamed. However, preliminary experiments with our corpus of Haskell programs suggests that this is not always a problem in practice.

Moss demands that programs be sent to the Moss server for comparison, and some assessors may not feel comfortable with such a situation (or their universities may in fact forbid it). 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 paper 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 the literature further.

Holmes was constructed in two steps: the second author implemented a large collection of heuristics for a subset of Haskell, Helium programs [7] to be precise. This has been documented in a master's thesis [12]. 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 assist in discovering plagiarism by sorting pairs of submissions or modules on observed 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 future work, we mention a comparison with the results provided by the web-accessible Moss [11]. Moreover, 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 [9].

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

Acknowledgements The authors thank Alexander Aiken, Jan Rochel, Stefan Holdermans, Doaitse Swierstra, Sean Leather, Atze Dijkstra, Arie Middelkoop, Marco Morazan, Johan Jeuring and Niklas Broberg for their help with this paper.

References

[1] A. Ahtiainen, S. Surakka & M. Rahikainen (2006): *Plaggie website*. http://www.cs.hut.fi/Software/Plaggie/.

[2] S. Clover (2008): *Haskell diff library*. `http://hackage.haskell.org/package/Diff`.

[3] M. R. Garey & D. S. Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman.

[4] D. Grune & M. Huntjens (1989): *Het detecteren van kopieën bij informatica-practica*. *Informatie (in Dutch)* 31(11), pp. 864 – 867. http://www.cs.vu.nl/ dick/sim.html.

[5] J. Hage (2006): *Programmeerplagiaatdetectie met Marble*. Technical Report UU-CS-2006-062, Department of Information and Computing Sciences, Utrecht University.

[6] J. Hage, P. Rademaker & N. van Vugt (2011): *Plagiarism detection for Java: a tool comparison*. In: *Computer Science Educaiton Research Conference*, CSERC '11, Open Universiteit, Heerlen, Open Univ., Heerlen, The Netherlands, The Netherlands, pp. 33–46. Available at `http://dl.acm.org/citation.cfm?id=2043594.2043597`.

[7] B. Heeren, D. Leijen & A. van IJzendoorn (2003): *Helium, for learning Haskell*. In: *ACM Sigplan 2003 Haskell Workshop*, ACM Press, New York, pp. 62 – 71.

[8] V. I. Levenshtein (1966): *Binary codes capable of correcting deletions, insertions, and reversals*. *Soviet Physics Doklady* 10(8), pp. 707–710.

[9] H. Li, C. Reinke & S. Thompson (2003): *Tool Support for Refactoring Functional Programs*. In Johan Jeuring, editor: *ACM SIGPLAN 2003 Haskell Workshop*, ACM, pp. 27–38. Available at `http://www.cs.kent.ac.uk/pubs/2003/1677`.

[10] L. Prechelt, G. Malpohl & M. Philippsen (2002): *Finding Plagiarisms among a Set of Programs with JPlag*. *J. of Universal Comp. Sci.* 8(11), pp. 1016 – 1038.

[11] S. Schleimer, D. Shawcross Wilkerson & A. Aiken (2003): *Winnowing: Local Algorithms for Document Fingerprinting*. In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ACM, pp. 76 – 85.

[12] B. Vermeer (2010): *Holmes, Hunting for Haskell Frauds*. Unpublished manuscript, `http://www.cs.uu.nl/people/jur/brianvermeer-msc.pdf`.

[13] M. J. Wise (1996): *YAP3: improved detection of similarities in computer program and other texts*. In J. Impagliazzo, E. S. Adams & K. J. Klee, editors: *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, 1996, Philadelphia, Pennsylvania, USA, February 15-17, 1996*, ACM, pp. 130–134. Available at `http://doi.acm.org/10.1145/236452.236525`.

A An example of plagiarised code

In 6 we display two pieces of code that are considered to be very similar by our tool. Note that the students have in fact tried to cover up the plagiarism, but that some parts give them away. The functions have been reordered (the ordering can be reconstructed by the assessor by looking at the types), many identifiers have been renamed, and the Dutch comments have been rewritten. For reasons of anonymity and space some of the code has been removed. The only modification made with respect to the original code was to break the second case of *vergelijk* into two lines, because the actual line of code was too long.

```
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]
```

```
  -- 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
```

Figure 6: Two refactored plagiarised pieces of code (from `fp-fql`, 2002)