Plagiarism Detection for Lisp

António Menezes Leitão
antonio.menezes.leitao@tecnico.ulisboa.pt
INESC-ID/Instituto Superior Técnico, University of Lisbon
Lisboa, Portugal

ABSTRACT

Computers made it very easy to copy someone else's work. This makes grading a difficult task, as the teacher that wants to prevent plagiarism needs to compare each student's assignment against every other student's assignment, a quadratic process that is impractical when the number of assignments gets large. Students know this and some take advantage of it. To be able to detect plagiarism among students' programming assignment we created a software tool that checks all assignments against each other, searching for copied fragments. Unlike many other tools, this search is not based on textual comparisons or hashing functions but, instead, on collecting pieces of evidence for and against a plagiarism verdict. This collection is determined by specialized procedures, invoked in a data-driven fashion, that incorporate expert knowledge regarding what is plagiarism and what is not plagiarism. The tool has been successfully used since 1995 in the evaluation of assignments programmed in Lisp dialects, particularly, Common Lisp, Scheme, Racket, and AutoLisp, and its mere existence became a deterrent for plagiarism.

ACM Reference Format:
António Menezes Leitão. 2019. Plagiarism Detection for Lisp. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Nowadays, every student uses a computer to accomplish his assignments. Unfortunately, the computer also makes it very easy to copy and share students' work. The computer even helps in camouflaging the copied parts so that the teacher does not easily detect them. This situation is called plagiarism [15] and it is based on making a series of systematic changes to a program to create a derivative work that is syntactically different but semantically identical. The problem is especially serious in Computer Science courses, where evaluation typically includes the development of software projects. In this case, it is very easy to make copies that, at first sight, look very different from the original. Besides, when the assignments to be graded are divided among teachers, it is almost impossible to detect the copies.

In this paper, we discuss a set of techniques for the detection of copies in students' projects and we present a software tool that implements those techniques for Lisp-based projects. The tool was evaluated in real cases of projects developed using Common Lisp, Scheme, Racket, and AutoLisp and the results confirm its ability to detect plagiarism, even when the projects suffered considerable changes in order to hide their non-originality.

In the next section, we discuss the detection of plagiarism. In the following sections, we present an approach for its automation and specific details of its implementation in a plagiarism-detection tool. In the Results section, we evaluate its use in a real case. The final section summarizes the work and compares it with other approaches.

2 DETECTING PLAGIARISM

The intent of a plagiarized work is to obtain a grade without being associated with the original work from which it was derived. This implies that a carefully plagiarized work presents sufficient differences from the original that the teacher cannot easily correlate them. Unfortunately, seldom there is time to make a really good plagiarized work and some traces of the original remain in the copy. This is particularly true in software, where the copy needs to preserve the semantics of the original algorithms. In this case, the copier can change some particular parts of the original but many other parts such as the global structure, the control structures, the recursive or iterative nature of some functions, or the primitive operations used, remain as traces of the original. It is these traces that help identify plagiarism.

We now describe the principles behind the proposed plagiarism detection tool. The tool is intended to detect plagiarism between programs written in Lisp dialects, namely, Common Lisp, Scheme, Racket, and AutoLisp. In all of these languages, a program can be subdivided into smaller and relatively independent parts. Depending on the particular programming language, these parts are named *classes*, *functions*, *procedures*, *structures*, *methods*, etc. In most high-level languages, and even more so, in Lisp dialects, there are few limitations upon the textual form of the code and so it is possible to interchange parts of the program as well as change its textual indentation without affecting its behavior. Additionally, the program is, to a large extent, independent of the particular names used to describe it. This means that two programs may be textually different but semantically identical.

Usually, the visual detection of copies in programs is based exclusively on its textual form. Students know this, and take advantage of it. By using all the possibilities described above, they can modify the textual form of a program is such convoluted ways that it no longer resembles the original. If they have enough care not to change the semantics of the original, the copy will accomplish the same task. This suggests that the detection of copies should not be based on the textual form of the programs but on its semantics.

Given that programs are composed by subprograms, and there is no order between the subprograms, in order to identify plagiarism

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'19, April 01–02 2019, Genova, Italy
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-2-9557474-3-8.

in two programs we have to compare all the subprograms in one program against all the subprograms in the other program. If these subprograms are basic units, i.e., they do not contain any subprograms, then they can be compared in terms of the control structures they use, the primitive operations upon which they depend, etc. If both subprograms start by using an `if` control structure, most people would agree that this, by itself, is not a verdict of plagiarism, but it can be considered as a very small piece of evidence. In order to collect more pieces of evidence, we check the test form of the control structure. If in both cases this form is the application of the very same primitive function then we increase our confidence that we are facing plagiarism, otherwise, we decrease it. Then, we look at the consequent of the control structure and we apply similar reasoning. If, for example, in both consequents the program execute a loop until some criterion is fulfilled, this is another piece of evidence. We proceed by comparing the termination criteria of both loops, as well as the body of the loops. In case the forms in both subprograms do not match, we search for some of the common tricks students use to transform one into the other, such as swapping arguments in commutative operations. This process proceeds until we exhaust both subprograms. In the end, we decide whether we collected enough evidence to signal the subprograms as suspicious.

The process just described is commonly done by teachers that grade students' projects. However, they tend to do it only if they already suspect that there was some cheating, which makes the process unfair, as some are detected and others are not. Moreover, the process does not scale, either in the amount of information that teachers can collect, or in the number of comparisons between projects that they can do. In this paper, we address these two problems, presenting a tool built specifically to detect plagiarism between Lisp-based student projects.

3 AUTOMATIC PLAGIARISM DETECTION

We now describe an approach and a tool for automatic detection of plagiarism in Lisp-based projects. We will use projects written in the Common Lisp language [22] as an example but the tool is configurable to operate with other Lisp dialects and, with some extra effort, with other non-Lisp languages.

The tool is, essentially, a function which accepts two programs as arguments and returns a similarity value. This value abstracts away the particular similarities between the compared programs and is related to the level of confidence that we have in a verdict of plagiarism.

The tool can be decomposed into four main components:

- A parser that builds a syntax tree describing the programs to compare. This allows us to forget the textual form of the programs.
- A collection of specialized compare functions, each one designed to analyze particular features of the code that is being compared and to compute its similarity value.
- A data-driven matcher that identifies particular pieces of code within the programs to compare and selects the specialized compare functions appropriate to those particular pieces of code. This way, the tool can be easily extended or adapted to work with other languages.
- A combination function that combines all the similarity values found by the previous modules into a global value.

The parser is the easiest part of the tool because there are already some available for most high-level languages. In fact, for the Common Lisp language we just use the plain `read` function.

After parsing the two programs to compare, we feed the syntax trees found into the compare function. Since we may have one specialized compare function for comparing `if` forms, and another specialized compare function for comparing iteration forms, and so on, the top-level compare function will then check which one of the specialized compare functions is appropriate to analyze the two programs. To this end, each specialized compare function includes two patterns, each one matching one particular form on each of the compared programs. When the patterns match, the specialized compare function uses pattern variables to access the sub-forms of the programs and recursively compares those sub-forms using the top-level compare function until we reach atomic pieces of code.

In each recursive step, we collect pieces of evidence suggesting that we are facing copied programs. These pieces of evidence have a numeric value which may be larger or smaller depending on their importance. As an example, if both compared programs use a rarely used primitive, this is higher evidence of plagiarism than if both use a more frequent operation.

Since the number of forms to compare is quite large and there may be a large number of specialized compare functions to choose from, it is important that each of the operations involved—selecting the compare function, matching its patterns, and combining the amount of evidence found—executes quickly. We now analyze each of these operations.

3.1 Evidence

During the compare process we collect pieces of evidence *for* and *against* a verdict of plagiarism. For instance, it is unlikely that a Lisp function that starts with an `if` special form can be considered a copy of another that starts with the primitive function `car`. It is the combination of all collected evidence—for and against—that leads to the final verdict.

Expressing and combining evidence is a problem that has already been treated by [20], in the context of diagnosing medical problems. In this area, almost nothing is certain and doctors can only gather evidence for diseases based on the patient's symptoms. For instance, if I have a headache, I may have got a cold, but it is also possible that I have one or more of many other diseases such as cancer, meningitis, etc. To identify the correct disease, doctors collect and combine other pieces of evidence, like my body temperature, or the fact that I sneeze or not.

In our case, we are only interested in determining if the plagiarism "disease" is present and there are pieces of positive evidence and pieces of negative evidence for this possibility. For instance, when comparing two functions, an equal number of parameters constitutes a (small but) positive piece of evidence that they are a copy of each other, while a different number of parameters constitutes a negative one. Following the work of [20], we define evidence as a number in the interval $[-1, 1]$. A value in this interval represents the amount of evidence that we have about something and is called *certainty factor*. A certainty factor of 1 means absolute

certainty that it is true, −1 means absolute certainty that it is false, and 0 means that we have nothing for or against—we simply do not know.

To combine evidence, we cannot simply add them, as we need to maintain the combined evidence within the interval $[-1, 1]$. Again, we refer to [20] to justify the following combination function C:

$$C(x, y) = \begin{cases} x + y - xy & \text{if } x > 0 \text{ and } y > 0, \\ x + y + xy & \text{if } x < 0 \text{ and } y < 0, \\ \frac{x+y}{1-\min(|x|,|y|)} & \text{otherwise.} \end{cases}$$

The parameters of the function are two certainty factors that we want to combine. The function combines then in such a way that combining true with something else except false is true, combining false with something else except true is false, combining unknown with something else does not change the result and combining true with false is an error as it would be equivalent to a contradiction. When two pieces of evidence are *for* something, that is, they are both positive, its combination is stronger evidence for that thing. When two pieces of evidence are *against* something, that is, they are both negative, its combination is stronger evidence against that thing. When one of them is positive and the other is negative, its combination will be something in between.

Although certainty factors have well-known problems regarding the dependency of evidence, the distinction between conflict and ignorance, and the undefined semantics of the certainty factor itself, they are simpler and computationally more tractable than other approaches such as probability theory [16], Dempster-Schafer theory [6], or fuzzy set theory [4].

As an example of the use of certainty factors in our tool, we now present the evidence defined for the specific situation of a Lisp conditional: the cond macro without a default clause.

```
(defevidence both-conds-miss-default 0.5 -0.3)
```

In the previous example, associated with the name of the evidence there are two values, one for the evidence in favor of plagiarism, and the other for the evidence against plagiarism. In this particular example, these evidences are relatively large because a cond without a default clause is a rare situation. When it occurs in the same place in two programs, it is a strong indicator of plagiarism. When it occurs in one program and not in the other, it is a medium indicator against plagiarism.

3.2 Defining Compare Functions

To be able to accurately compare Lisp forms, we must specify different comparing functions for different combinations of Lisp forms. In this section, we deal with the problem of specifying the *form* of Lisp forms.

Our idea is to use an association between patterns describing Lisp forms and expressions computing the evidence of a copy between those Lisp forms. The patterns will be matched against some form and, when successful, pattern variables will be bound to selected elements in the form. Then, the sub-forms designated by the pattern variables are compared and the pieces of evidence found are combined.

As an example of a specialized compare function, consider the situation where we have one `if` special form on each compared program:

```
(defcompare ((if ?test1 ?conseq1 ?altern1)
             (if ?test2 ?conseq2 ?altern2))
  (combine-evidence
    (get-evidence both-ifs t)
    (compare ?test1 ?test2))
    (compare ?conseq1 ?conseq2)
    (compare ?altern1 ?altern2))
```

The macro `defcompare` uses the common practice of tagging pattern variables with a leading question mark to distinguish them from other literal symbols. Note also that the body of the specialized compare function calls the generic compare function to compute the evidence for copy of the test, consequent, and alternative of both `ifs` and combine the returned values with the specific evidence of having two `ifs` on both programs.

Unfortunately, the previous comparison is too strict and does not search for additional signs of plagiarism. To that end, the actual comparison function is the following:

```
(defcompare ((if ?test1 ?conseq1 . ?altern1)
             (if ?test2 ?conseq2 . ?altern2))
  (combine-evidence
    (get-evidence both-ifs t)
    (compare-if-args ?test1 ?test2
                     ?conseq1 ?conseq2
                     (if ?altern1 (first ?altern1) nil)
                     (if ?altern2 (first ?altern2) nil))
    (get-evidence both-ifs-as-whens
      (and (null ?altern1) (null ?altern2)))))

(defun compare-if-args
    (test1 test2 conseq1 conseq2 altern1 altern2)
  (max (combine-evidence
        (compare test1 test2)
        (compare conseq1 conseq2)
        (compare altern1 altern2))
       (combine-evidence
        (compare test1 `(not ,test2))
        (compare conseq1 altern2)
        (compare altern1 conseq2))))
```

Note that, besides recognizing the case where the `if` is being used as a when, this improved comparison also verifies if the consequent and alternative might have been swapped.

The comparison function is opportunistically called with two forms and it will then try to match its patterns against those forms and, in case of success, bind the variables to the correspondent matched sub-forms. In case one of the patterns does not match, the function immediately returns with a null result.

Usually, the result of a successful match is a substitution list where pattern variables get their bindings. Unfortunately, there are two problems with this solution:

- The match is expensive. The result of the match is a structure that consumes time and space (that becomes garbage very soon). As the match is a fundamental operation there should be a minimum of garbage involved.
- There isn't a direct connection between the bindings found by the match process and the free variables in the compare function body.

Both problems can be solved if we take into account that the patterns are known at compile time. Thus, there is no need to

depend upon a generic match algorithm. We can optimize every match process because we know in advance what it must do.

3.3 Partial Evaluation

The kind of optimization that we are talking about is named *partial evaluation* and is a technique that aims at speeding up a program by specializing it on some of its inputs.

In generic terms, if we have a program F that expects an input i, we define the execution of that program as the result of $F(i)$. Now, let us consider that the input i can be split into two parts, one *static*—known in advance—and the other *dynamic*. Let us write the execution of our program on these inputs as $F(s, d)$ where s and d are the static and the dynamic parts of the input i, respectively. Since we know the static part of the input, it may be possible to rewrite the program F so that all computations that depend upon the static part of the input are already done, that is, we want to write a new program F_s such that $F(s, d) = F_s(d)$.

The program $F_s(d)$ is named the *residual program* for F with respect to s or, equivalently, a version of F specialized to s.

A *partial evaluator* is a program that specializes other programs with respect to some static input. Historically, a partial evaluator is known as *mix* (after [9]). A partial evaluator is, then, a program mix such that for every program F and static input s, $\text{mix}(F, s) = F_s$.

In our case, the program F is the match algorithm, and the static input s which we want to use to specialize the program is the known pattern. The result of the partial evaluation is a specialized function that no longer receives the pattern but is capable of implicitly matching it. To do this, we write a partial evaluator $\text{mix}_{\text{match}}$ already specialized on the match program. We then use $\text{mix}_{\text{match}}$ on some pattern to produce a specialized matcher for that pattern: $\text{mix}_{\text{match}}(s) = \text{match}_s$

Although there are already many partial evaluators available (for example [10], [17], and [19]), we decided to build our own. This decision was made because our patterns are quite simple, allowing a specialized partial evaluator to generate very fast code. Besides, our partial evaluator generates code which is integrated with the evaluation of evidence. The result of our partial evaluation of a compare function is another function that interleaves the matching and binding processes until all patterns are matched, and then evaluates the compare function body in the lexical environment established by the binding process.

3.4 A Data-Driven Approach

Since there are many kinds of Lisp forms, we also have many specialized compare functions. These functions must be invoked by a generic compare function when and only when they are needed. Since we want to be able to add more specialized compare functions in the future, we need some way of doing it without disturbing the rest of the compare functions. The *data-driven* programming methodology [13] is an appropriate solution.

In a data-driven approach to our compare problem, we keep all specific compare functions on a database. This database is dynamic in the sense that new specific compare functions can be added at any time, even during the compare process. The generic compare function will use this database to identify the specific compare function appropriate for each situation. In many cases, there will be more than one specific compare function matching a particular situation. For example, if we consider the Lisp expression `(+ 1 2)`, we can easily build patterns that match it, such as `(+ 1 ?arg)`, `(+ ?arg1 ?arg2)`, `(?f ?arg1 ?arg2)`, `(?f . ?args)`, `(?car . ?cdr)`, or simply `?expr`. All these patterns may coexist in our system because they have different purposes: `(+ 1 ?arg)` matches an increment operation, `(+ ?arg1 ?arg2)` matches a sum operation, `(?f ?arg1 ?arg2)` matches a two-argument function application, `(?f . ?args)` matches a general function application, and so on. To decide which compare function should be used, there must be an order between all the patterns. This order is needed to ensure that more specific compare functions are tested before more generic compare functions.

There are two perspectives regarding this order:

- Automatic-based: If we can compute the specificity of a pattern, we can use it to sort the compare functions so that more specific patterns are tried before less specific patterns.
- Manual-based: Since text files are sequential, there is a sequential order in the compare function definitions. We can arrange our definitions so that their relative position reflects our intended matching order.

The first approach is similar to what is used in *CLOS*—The Common Lisp Object System [11, 22], where multi-methods can be specialized on the type of all arguments. These types have a subtype relation between them that must be taken into account when dispatching a generic function. In our case, we have patterns with different degrees of specificity and we also have multi-patterns, thus being comparable to CLOS. The problem with this approach is that it is not always obvious which patterns are more specific and thus, we might end up with the wrong order.

Depending exclusively on the second approach is also somewhat problematic. Whenever we want to add some new patterns, we need to carefully choose where to define them. If we define them after more generic patterns, the new patterns will never be checked. If we define them before more specific patterns, the older patterns will never be checked.

Our solution to the problem is a mixed approach. We will retain the flexibility of the CLOS approach while depending on the definition order in situations where a generic dispatch would be ambiguous. The idea is to define a subsumption relation between patterns, and use that relation to order the patterns. Whenever the subsumption relation cannot decide which pattern is more specific, we use the definition order. This approach follows the programmer's intention except when his intention leads to wrong results.

3.5 Indexing

When the number of compare functions is large, checking the patterns of all compare functions until one of them succeeds is very time-consuming. Besides, this time grows as we add more compare functions. To solve this problem, when patterns contain literals, they are used as indexes in an hash-table of compare functions. Since each compare function specifies two patterns, we have a doubly-indexed hash-table. On each entry of the hash-table, we keep all the compare functions whose patterns begin with the correspondent keys, sorted by the specificity of the rest of both patterns. This allows a very fast search of the appropriate compare function in

most cases. In other cases, the indexing mechanism restricts the subset of applicable compare functions to just two or three, which is still quite good.

It is important to note that the indexing mechanism is itself implemented by a specialized compare function. This function has a parameter list which matches all patterns that can be indexed. When the match succeeds, the function uses the literals found to get the appropriate compare functions. This allows the indexing mechanism to be included in the system without modifying it. This is useful because different programming languages may require different indexing mechanisms and we can have several indexing mechanisms at the same time.

With this technique, our generic compare function needs only to check each specialized compare function in turn, according to the subsumption relation between the patterns of those functions. Some of these specialized compare functions are in fact indexing functions which speed up the selection of the appropriate compare functions. In case they are not applicable, the process continues, checking another function, being it a real compare function or just another indexing function. Obviously, it is possible to have indexing functions within indexing functions without limit. An indexing function can thus be seen as an abstraction of several compare functions.

4 COMPARING PROJECTS

The previous sections described the techniques that we developed for a generic compare process. We now describe some of the specific enhancements that specialize the process for the Lisp language.

4.1 Comparing Project Structure

Let us suppose we are comparing the following Lisp project:

```
(defun a (x y)
  (b x)
  (c y))
(defun b (z)
  (c z))
(defun c (z)
  z)
```

and the following copy:

```
(defun b1 (z1)
  (c1 z1))
(defun a1 (x1 y1)
  (b1 x1)
  (c1 y1))
(defun c1 (z1)
  z1)
```

The functions `a`, `a1`, `b`, `b1`, `c`, and `c1` are so small that they can hardly be considered copies. Nevertheless, there are obvious resemblances between the two projects, as they share the same structure. In the previous example, `a` calls `b` and `c` while `a1` calls `b1` and `c1`.

In order to compare the projects' structures, a possible solution would be to generate the callers/callees graph of both projects and compare them with a graph comparing algorithm. However, there is a simpler solution: whenever we find functions applications, we check their definitions just as we are checking the current ones.

Using the previous example, while comparing `a` and `a1` we find that they call, respectively `b` and `b1`. Then, we compare `b` and `b1` just to find that they call `c` and `c1`. Again, we compare `c` and `c1`. The result of the comparison is then returned and combined with the comparison of `b` and `b1`, and the result is returned and combined with the comparison of `a` and `a1`. This way, functions near the root of the graph will see its comparison getting more and more precise as each called function is compared.

This process also takes into account self-recursive and mutually-recursive functions. When in presence of such cases, we stop the check (avoiding infinite regression) and we return with a value reflecting that additional piece of evidence.

4.2 Comparing Lisp Forms

As we said before, when we compare two Lisp forms we first try to use specialized compare functions for those forms. The following list is just a short extract of some cases that the specialized compare functions deal with:

- To compare two functions, we check their names, their arguments lists, their documentation strings, and their bodies.
- Common transformations involving `let` forms consist of exchanging some bindings, and replace `let` with `let*` and vice-versa. This forces us to compare `let` forms, `let*` forms, and combinations of them.
- Another common transformation consists of cascading `lets`. Instead of using a single `let` to establish bindings, it is possible to use a large number of `lets`, each one establishing some of the bindings. Although the result is semantically similar, syntactically it looks very different. To avoid this trick, we collect all subsequent lets. Although we may be changing the semantics of the form, the change is not relevant to the compare process.
- While comparing bindings, we must be careful about unusual bindings. Although the usual `let` form is similar to `(let ((var val)) body)`, it is possible to have uninitialized bindings, such as `(let ((var)) body)`, or unique uninitialized binding such as `(let (var) body)`.
- When comparing `ifs`, we give special attention to the exceptional cases of `ifs` without alternative because they are rare. In this case, the `ifs` look like `whens`.
- When an `if` is copied, most students are smart enough to modify it by negating the test and swapping the consequent with the alternative. To deal with this case, we must also compare the possible transformation. We return the highest value found.
- Since the macro `cond` is very common, it is difficult to depend on it to detect copies. For this reason, we just dispatch on the tests and actions. However, we do analyze the default clause because its absence is a rare situation. If two `cond` forms both miss the default clause, that is a strong sign that they may have been copied.
- One of the common copying practices is replacing a `cond` by an `if` and vice-versa. To handle this case, we compare the first `cond` clause with the condition and consequent of the `if` and we compare a new `cond` containing the remaining clauses with the alternative of the `if`. This approach allows the comparison of a multi-clause `cond` with a cascade of `if` forms.

- Transforming between an `if` and `when` is another common plagiarism technique. The only tricky situations are (1) that a `when` with multiple consequents implies an `if` with a `progn` consequent and (2) that the `if` should not have an alternative or it should be `nil`.
- Another common transformation is between `if` and `unless`. As with the `if` and `when` comparison, the only tricky situations are (1) that an `unless` with multiple consequents implies an `if` with a `progn` consequent and (2) that the `if` should not have an alternative or it should be `nil`. Besides, the `if` must have a negated test or have the consequent swapped with the alternative.
- Substituting a `dotimes` with a `do` and vice-versa are also good approaches to hide plagiarism. To detect these situations, we compare the corresponding parts, also including the `dotimes` macro-generated ones (namely, stopping condition for the loop and increment expression).
- Comparing `setqs` and `setfs` needs to take into account that each variable must have its value, but the order between variable-value pairs can be exchanged (to a certain extent).

4.3 Comparing Transformations

We have already exposed some of the common copying techniques explored by students. Some of these techniques are sufficiently general to be easily formalized. On this set, we include changing parameter names, permuting function arguments, and cascading `let` bindings.

However, there is another set of techniques that are much more elaborated and involve more knowledge about Lisp functions. Transforming `(1+ expr)` into `(+ expr 1)` is an example. They are semantically equivalent expressions and either can be used. Since they are syntactically different, they make copy detection a harder task. There are much more examples, for instance, `(null expr)` and `(not expr)`.

Different Lisp functions that perform very similar operations in most contexts are another source of problems. The functions `car` and `first` are equivalent and either can be used. The functions `endp` and `null` have similar semantics and in most situations either can be used.

We extended our tool with some syntactic sugar to make it easy to define new compare functions to recognize such equivalent forms. Below we present some of the defined transformations.

```
(1+ ?x) ≡ (+ ?x 1)
(1- ?x) ≡ (- ?x 1)
(< ?x ?y) ≡ (>= ?y ?x)
(> ?x ?y) ≡ (<= ?y ?x)
(null ?x) ≡ (eql ?x nil)
(zerop ?x) ≡ (= ?x 0)
(car ?x) ≡ (first ?x)
(cdr ?x) ≡ (rest ?x)
(null ?x) ≡ (endp ?x)
(= (length ?list) 1) ≡ (endp (rest ?list))
(not (null ?list)) ≡ ?list
(cons ?elem nil) ≡ (list ?elem)
```

As is visible, we describe the equivalences using repeated pattern variables in both patterns. Usually, when a pattern variable is repeated on two patterns, it means that for them to match, the variables' values must be equal. Here, it just means that we should compare the values on each matched form. When we enter the equivalent forms into the system, all pattern variables are renamed and then two new compare functions with permuted arguments are defined so that both transformations can be tried.

In order to define these transformations, the macro `defequivs` takes a list of equivalent forms and, optionally, the corresponding evidence. As an example, the form

```
(defequivs ((cadr ?x) (car (cdr ?x)) (second ?x) (nth 1 ?x)))
```

creates 12 comparing functions, each dealing with one of the two-element combinations of the given patterns.

5 RESULTS

Validation is an important part of the plagiarism detection process. To this end, we also developed a mode for the Emacs editor [21] that simplifies the verification of the results. The mode presents, side by side, the fragments of code that were considered sufficiently similar to merit a careful observation.

We tested our tool on a course where students had to create a moderately complex project in Common Lisp. One of the requirements was that students had to implement a small number of functions with a pre-established signature.

The students submitted 112 projects with an average of 27 implemented functions per project. We conducted tests where we compared just a selected function and tests where we compared all the implemented functions. On the first case, there were 6216 comparisons between projects. On the second case, the number raises to 4.4 million.

5.1 Analyzing a Selected Function

In this test, we analyzed a selected function that all students had to implement. The results are in Figure 1, where each axis represents the project's number and each square has a shade that is proportional to the amount of evidence found. In this figure, we note the selectivity of the compare process. Although all projects defined the selected function, only a few pairs (16/6216 = 0.26%) were found suspicious. This means that originality was high, that is, there were many different implementations of the selected function. On the other hand, if we count the number of suspicious projects (not *pairs* of projects), the ratio grows to 21/112 = 18.8%. Note that the first percentage represents the number of comparisons that were found positive, while the second percentage represents the affected projects. The first number will only be 100% when all students use the very same function, while the second can reach that value if there is one copy for each original function.

To evaluate the tool's accuracy, all suspicious projects were manually checked. We also checked projects with small negative evidence to be sure that the tool did not miss anything. This was important to determine an evidence level which justifies further inspection. This value depends on the average size of the functions, the difficulty of the assignment, etc. In our case, we found that evidence below 0.6 does not represent plagiarized functions but, instead, functions that happen to be similar.

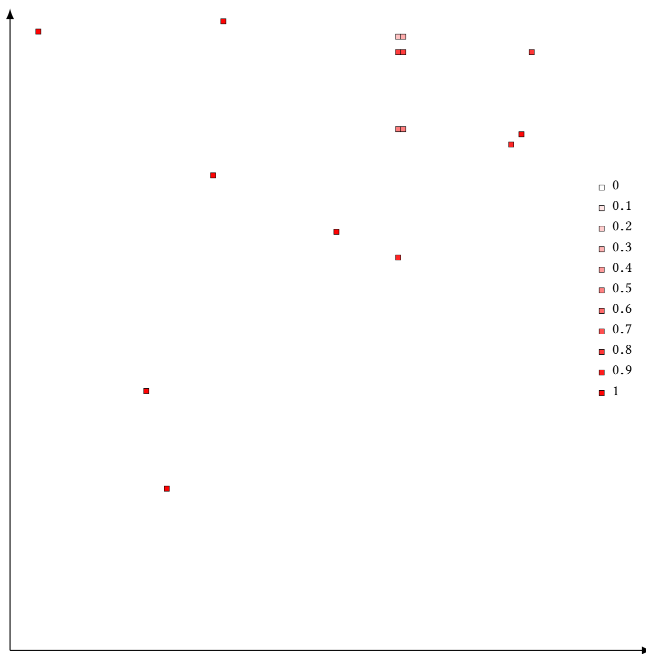

Figure 1: Evidence of copy of a selected function between all pairs of projects. Both axis represent the set of compared functions.

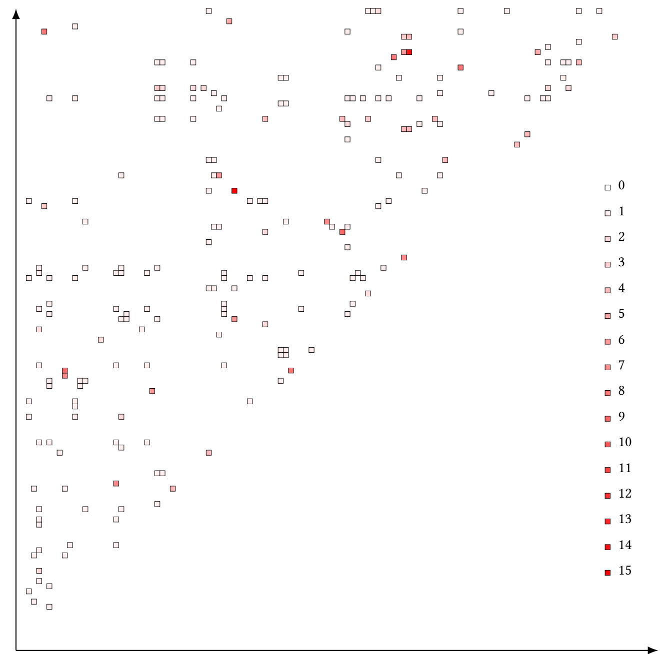

Figure 2: Number of copies among pairs of projects. Both axis represent the set of compared projects.

5.2 Analyzing Projects

An unrestricted test involving all projects' definitions can produce a very large number of suspicious pairs of definitions. To avoid cluttering the results with irrelevant information, we only searched for very strong evidence of copy (larger than 0.8). Note that we excluded from this test the critical functions tested by the previous analysis.

Figure 2 represents the results of the analysis on pairs of projects. A combined analysis of the previous two figures shows that there are data points in Figure 2 that are not present in Figure 1. This means that there are students who are careful enough to develop the critical parts of the project but careless in what regards less relevant parts.

In the end, the final number of copied projects detected was 28, which represents 25% of all projects, the highest ever recorded on that course.

5.3 Pedagogical Effects

Besides helping teachers detecting plagiarism, the tool is invaluable as a pedagogical measure. The mere fact of knowing that assignments will be scrutinized regarding possible plagiarism entails in the student a completely different attitude.

To verify the change in students' habits, we run the tool again one year later, on the very same course. This time, the project was given to 132 groups of students and 109 of them delivered a solution.

We present in Figure 3 the results of the plagiarism detection process. As before, only pairs of projects with more than 2 copied forms are shown. As we can see, there are much less copied projects now.

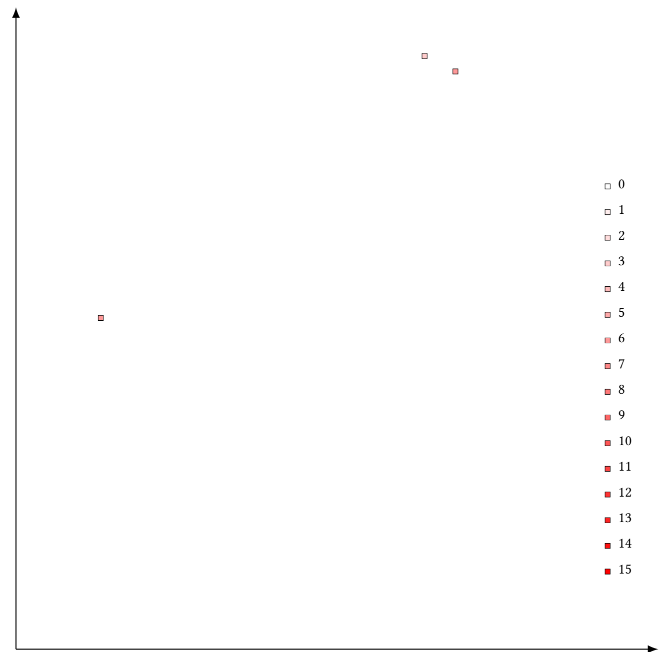

Figure 3: Number of copies among pairs of projects in the following year. Both axis represent the set of compared projects.

In our opinion, the main reason for these results is the psychological effect of the perceived increase in the plagiarism detection effort. Independently of the students' suspicions regarding the real cause of that increase, they quickly adapt to this perception by either avoiding plagiarism or by increasing the effort to modify the plagiarized projects so that cheating would not be detected.

6 RELATED WORK

The plagiarism detection tool here presented was invented in 1995. Despite being frequently used in the following years, it remained a closely guarded secret because we were afraid that it would not be well-received by the student's association. Nowadays, when automatic grading is already a given in universities, plagiarism detection would not surprise anyone but, at that time, that was not the case. In fact, the existence of the tool was acknowledged only after several other similar tools were announced.

Plagiarism detection was studied by several authors, including [3, 5, 7, 12, 14]. The proposed approaches include textual comparison [1], that compares source code modulo name changes, abstract syntax tree (AST) comparison [2, 23, 24], which is immune to textual changes but does not detect code transformations, metric analysis [5, 7], that detect code framents that have a similar number of unique operators, operands, declared variables, etc., and fingerprinting [8, 18], based on the use of hashing functions that are immune to the typical code transformations done to hide plagiarism, so that two plagiarised fragments produce the same hash value.

The work here presented can be considered as an extended AST-based comparison approach that is immune to the code transformations generally employed to hide plagiarism, and where the similarities and differences between AST nodes are evaluated using a metric-based approach. The metrics were derived from our experience regarding the code that is typically written by our students. A final difference is the adaptability and extensibility of our approach, which was designed to easily support user-defined comparisons and metrics.

7 CONCLUSION

We described a tool to identify plagiarism in students' projects. The tool compares program fragments and gives a certainty factor to the question "is plagiarism present in these fragments?". Based on this certainty factor, the teacher can quickly identify students' projects that deserve more careful attention.

We have used the tool on a course and we found a surprisingly high number of copied projects. However, the results obtained in the following years show that students quickly adapt to the increased scrutiny by dramatically reducing plagiarism.

There are other important uses for this tool. In particular, it can be used to detect redundancies in code. By comparing the code against itself one can detect which code fragments are sufficiently similar to deserve being substituted by a conveniently parameterized function.

8 ACKNOWLEDGMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2019.

REFERENCES

[1] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In Linda M. Wills, Philip Newcomb, and Elliot J. Chikofsky, editors, *Proceedings: Second Working Conference on Reverse Engineering*, pages 86–95. IEEE Computer Society Press, 1995. ISBN 0-8186-7111-4.

[2] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In T. M. Koshgoftaar and K. Bennett, editors, *Proceedings; International Conference on Software Maintenance*, pages 368–378. IEEE Computer Society Press, 1998. ISBN 0-7803-5255-6, 0-8186-8779-7, 0-8186-8795-9.

[3] H. L. Berghel and D. L. Sallach. Measurements of program similarity in identical task environments. *ACM SIGPLAN Notices*, 19(8):65–76, August 1984. ISSN 0362-1340.

[4] Didier Dubois and Henri Prade. An introduction to possibilistic and fuzzy logics. In et al Smets, editor, *Non-Standard Logics for Automated Reasoning*. Academic Press, 1988. Reprinted in Readings in Uncertain Reasoning.

[5] J. A. W. Faidhi and S. K. Robinson. An empirical approach for detecting program similarity within a university programming environment. *Computers and Education*, 11(1):11–19, 1987.

[6] Jean Gordon and Edward H. Shortliffe. The dempster-shafer theory fo evidence. In Bruce G. Buchanan and Edward H. Shortliffe, editors, *Rule-Based Expert Systems*, pages 272–292. Addison Wesley Publishing Company, Reading, Massachusetts, 1984.

[7] S. Grier. A Tool that Detects Plagiarism in PASCAL Programs. *SIGSCE Bulletin*, 13(1), 1981.

[8] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of CASCON '92,* (Toronto, Ontario; November 9-11, 1992), pages 171–183, November, 1992.

[9] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Springer-Verlag, 1985.

[10] J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In S. L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 177–195. Berlin: Springer-Verlag, 1991.

[11] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A programmer's guide to CLOS.* Addison-Wesley Publishing Company, Cambridge, MA, 1989.

[12] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. In *Proc. 18th NIST-NCSC National Information Systems Security Conference*, pages 514–524, 1995.

[13] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, San Mateo, California, 1992.

[14] K. Ottenstein. An algorithmic approach to the detection and prevention of plagiarism. *ACM SIGSCE Bulletin*, 8(4):30–41, 1976.

[15] A. Parker and J. Hamblen. Computer algorithms for plagiarism detection. *IEEE Transactions on Education*, 32(2):94–99, May 1989.

[16] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, California, 1988.

[17] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In M Billaud, P Castéran, MM Corsini, K Musumbu, and A Rauzy, editors, *WSA '92—Workshop on Static Analysis*, number 81-82 in bigre, pages 109–117, Bordeaux (France), September 1992.

[18] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[19] P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Berlin: Springer-Verlag, 1996.

[20] E. H. Shortliffe. *MYCIN: A Rule-Based Computer Program for Advising Physicians Regarding Antimicrobial Therapy Selection.* PhD thesis, Stanford Artificial Intelligence Laboratory, Stanford, CA, October 1974.

[21] Richard M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. Technical Report AIM-519A, Massachusetts Institute of Technology, June 1979. URL ftp://publications.ai.mit.edu/ai-publications/500-999/AIM-519A.ps.

[22] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, second edition edition, 89. URL ftp://cambridge.apple.com/pub/CLTL/CLTL.tar.gz.

[23] Wise. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 24, 1992.

[24] Wise. YAP3: Improved detection of similarities in computer program and other texts. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 28, 1996.