Walker: Automated Assessment of Haskell Code using Syntax Tree Analysis

Rick de Vries
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
r.h.devries@student.utwente.nl

ABSTRACT

Programming educators often require students to use specific language features to ensure that they meet the educational goals. Verifying such requirements can be very time-consuming for teaching staff. This research investigates the usage of (static) syntax tree analysis to automatically validate the presence of required language constructs in Haskell programs.
This paper shows the effectiveness of this approach by testing a prototype written in Haskell (named Walker) on submissions by students, and discusses the different techniques used for traversing the syntax tree when validating the requirements. The results show the approach to be highly accurate, only showing weaknesses when evaluating student-defined types or deviating function names.

Keywords

Haskell, Functional Programming, Syntax Trees, Automated Assessment, Static Analysis

1. INTRODUCTION

Learning a new programming language can be a challenge to novice programmers, especially when this new language is in a new "paradigm" of programming. Haskell is often used as an introduction to the paradigm of Functional Programming, and features some unique language constructs that new programmers need to familiarize with.

While doing exercises, students are forced to use specific language constructs to help them adapt to the new style of programming. Unfortunately, many students do not read these exercises carefully, causing teachers and teaching assistants to have to manually check the code for usage of compulsory language features.

It would help all parties to have a way to check the student solutions for adhering to the exercises in an automated way. However, this is not as trivial as it might seem: simply comparing text to model solutions does not work for most programming exercises, including those for Haskell. Student solutions need to be checked for usage of the constructs, not for having an answer "similar enough" to the model solution. A solution for this could be an analysis of the syntax tree, which includes information about the language features used.

Permission to make digital or hard copies of all or part 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. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2019, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

In this paper, we will investigate such use of syntax tree analysis of Haskell programs for the purpose of automatic validation of language feature requirements. We will do so by the creation of a prototype, which we will test on submissions from first- and second-year students from the University of Twente.

We will discuss the different challenges that we faced when building Walker and the solutions we used to solve them, followed by the results that we obtained from our tests on real-world submissions. Before that, Section 2 discusses the various aspects that need to be assessed, after which Section 3 describes the existing automated grading solutions for Haskell and syntax tree-based approaches for education in general. Sections 4 and 5 then describe the different challenges that were encountered when building the prototype. The results of the testing are described in Section 6 and discussed in Section 7, from which the conclusions are drawn in Section 8.

2. REQUIREMENTS

The automated assessment should ignore the irrelevant details of the students' source code: as long as the (educational) requirements set out by the teacher are met somewhere in the solution, the submission should be accepted.

This means that students have the freedom to structure their answer as long as they satisfy those requirements. It should, for example, be possible for students to meet the criteria via (global) helper functions, locally defined helpers in a `where`-clause or by putting them in a nested expression (e.g., an argument to another function). In addition, students should be allowed to use pattern guards.

The assessment criteria should allow for nesting (e.g., a lambda expression as an argument to a map), and basic logical aggregators. At least the following language patterns for the Haskell programming language should be recognized:

- Use of list comprehension and Monads (and assess inner statements)
- Use of lambda abstraction (and assess inner expressions)
- Use of recursion
- Use of pattern matches on specific data constructors
- Use of specific (standard library) functions (such as `foldr` or `map`) (and assess arguments given)

In addition, the application should check if the student applied the desired type signature, as specified by the grader. The type signature could be polymorphic.

The input and output should be easily parseable by external applications, such as script to process the results into a spreadsheet or upload them to an online grading portal.

It should be noted that the correctness of the solution does not need to be tested: other tools already exist for this purpose, such as the lightweight QuickCheck tool [2]. The requirements are solely for the purpose of checking the structure of the solution, *not* its correctness.

3. EXISTING SOLUTIONS

Automated code assessment is a well-researched field, consisting of many different approaches for different languages and paradigms [7]. In addition, many researchers have looked into the structural analysis of source code to measure code similarity. Both purposes and their relevances are discussed in this section.

3.1 Automated assessment

The majority of tools for automated assessment rely on automated testing, program transformations or "basic static analysis", including calculations of the cyclomatic complexity or the presence of code structures [7]. This can be achieved in varying ways, ranging from bytecode analysis to syntax graph traversal [12].

For Haskell, Jeuring et al. developed the Ask-Elle system [5]. It uses program transformations to assess a submission. Program transformations attempt to normalize the source code of a program, while retaining the behavior. Some examples of program transformations include:

- the removal of redundant parenthesis;
- the standardization of variable names (alpha-conversion);
- inlining of values in a `let`-clause or a `where`-clause;
- replacement of equivalent function calls (e.g., replacing `drop 1` by `tail`).

Ask-Elle accepts a submission if it matches the model solution after applying the available program transformations. As a consequence, a solution will be rejected if the program transformations are insufficient. Besides that, the inclusion of (correct) type signatures by the student causes the tool to reject the solution.

Another noteworthy product is a submission system for a MOOC in OCaml, which provides a grading library with syntax tree traversal [1]. However, this software cannot readily be used for Haskell, as OCaml is substantially different from Haskell in a number of ways. For example, OCaml supports imperative constructs (e.g., loops), but does not support list comprehension or type constructor polymorphism[1].

3.2 Code similarity

While code similarity software obviously cannot be used to assess language feature usage, the underlying techniques can provide insights in different approaches for static code analysis. Code similarity software is often used for aiding in the detection of plagiarism. Different approaches are used to abstract from the superficial changes that students make to mask plagiarism attempts, such as the `diff`-tool in in Unix or more sophisticated token stream analysis [4].

[1]Lack of type constructor polymorphism causes all type variables to be restricted to kind "`*`" in OCaml, whereas in Haskell they can be of other kinds (e.g., "`* -> *`") and applied to each other.

In addition, different graph structures have been used for plagiarism detection, including call graphs, dependency graphs, control flow graphs and syntax trees [10]. For Haskell, the Holmes tool was developed, which relies on analyzing call graphs [9], token streams and document fingerprinting to detect code similarity [3].

4. DESIGN AND MODELLING

We made several design choices for the implementation of Walker. The first question was the language that the tool should be developed in (Section 4.1). Another consideration was the modelling of the code requirements (assessment criteria) as outlined in Section 2, to allow a teacher to specify what is expected from the students (Section 4.2). Finally, a generalized function model was necessary due to the many ways in which functions can appear in Haskell (Section 4.3).

4.1 Implementation language

In order to work on syntax trees, the code of the student solution needs to be parsed first. As such, it is convenient to write the tool in a language that has library support for this purpose. In addition, there should be some form of (de)serialization support to import and export the grading criteria.

For these reasons, Haskell is used as the language of choice. The availability of the *haskell-src-exts* and the *aeson* libraries fulfilled the requirements of parsing Haskell code and JSON (de)serialization, respectively. There were not many alternatives for the Haskell parser, partly due to Haskell being a context-sensitive language, complicating the creations of parsers.

4.2 Requirements specification

The requirement models should be able to specify the assessment criteria as listed in Section 2. Due to the nested nature of the criteria (for example, being able to specify constraints on arguments to specific functions), a nested data structure was necessary. The models used can be found in Listing 1 below.

```haskell
data LogicOp =  And | Or | Not
data Req = EmptyReq {
} | CombinedReq {
      reqOptions :: [Req],
      reqOp :: LogicOp
} | Recursive {
      isRecursive :: Bool
} | ReqTypeSig {
      typeSig :: String
} | FuncUsage {
      funcUsageName :: String,
      selfDefined :: Bool,
      argsExpr :: Req
} | LambdaFunc {
      numArgs :: Int,
      innerLambdaExp :: Req
} | ListCompr {
      innerListComprExp :: Req
} | MonadExp {
      innerMonadExp :: Req
} | PatMatch {
      constructorName :: String
}
```

Listing 1. Requirements models

Requirements can be combined using logic operations at any nesting level, since `CombinedReq` is a requirement in itself. For example, it is possible to specify that a predicate in a list comprehension expression should use either the "<"-operator *or* the ">"-operator. This can be achieved by using a `CombinedReq` (itself consisting of two `FuncUsage` instances and the `Or` operation) inside a `ListCompr`.

4.3 Function abstraction

Now that the requirements can be communicated to Walker, they need to be validated on the syntax tree of the submission. However, the syntax trees as produced by *haskell-src-exts* tend to be quite large. In addition, functions appear in many (syntactical) forms and are consequently represented differently in the parse tree.

We abstracted away from these syntactical differences in Walker, and represented all different forms of functions in one data class that we use instead of the complete syntax tree as produced by *haskell-src-exts*.

In particular, the following complicating factors of functions and syntax tree nodes from *haskell-src-exts* should be noted:

- When parsing a top-level function, different grammar rules and non-terminals are applied depending on the number of arguments. When there are no arguments, the function is parsed as a `PatBind`. When arguments are present, the syntax tree node is a `FunBind`.

- Functions do not always have a "single" expression as a result. Consider the use of pattern guards: the outcome of the function can be any expression on the right-hand side.

- Functions can be referenced in different ways:
  - Lambda functions do not have a name: they are anonymous, and often given as arguments to other functions.
  - Top-level functions have a single name, followed by their arguments.
  - Pattern bindings can have multiple identifiers that force their evaluation. For example, `(x:xs) = [1..10]` is evaluated when either `x` or `xs` is required.

- Functions can have where-clauses, which themselves can be functions or pattern bindings.

The following (recursive) generalizing class is used in Walker to overcome these issues.

```
data Func = Func {
    funcActivations :: [String],
    funcArgs :: [Pat'],
    funcRhss :: [Exp'],
    funcBinds :: [Func],
    funcReqs :: [Req]
} deriving (Show, Eq)
```

Listing 2. Generalized function model

The `Pat'` and `Exp'` types are shorthand type synonyms for the `Pat l` and `Exp l` classes from *haskell-src-exts* with predefined type arguments. More importantly, this generalization is sufficient to solve the aforementioned issues:

- `funcActivations` generalizes the different call methods for function bindings, pattern bindings and lambda functions by having different (number of) names in the list.

- `funcArgs` also allows the function to have zero or more arguments by virtue of being a list.

- `funcRhss` allows for pattern guards or having a "normal" right-hand sides.

- `funcBinds` contains the different bindings of a `where`-clause , and allows for being either a function or a pattern binding by being a `Func` itself.

- `funcReqs` is not necessary for abstraction, but for bookkeeping purposes. It allows Walker to differentiate assessed and non-assessed functions when traversing the syntax tree.

Together, these properties generalize the different function syntaxes, but still provide sufficient information to process the requirements while traversing the functions.

5. SYNTAX TREE PROCESSING

After Walker has parsed the students' code and converted the different functions into `Func` instances, the requirements are ready to be processed.

5.1 Scoping

Scope management is a vital part of Walker, although its need may not be immediately obvious. At first glance, it might seem sufficient to perform a search for a specific identifier (e.g., `map`) when usage of that function is required. To see that this is insufficient, consider the following workarounds for "using" the `map`-function to square all numbers in a list.

```
square [] = []
square (x:xs) = map x : square xs
    where map a = a^2

square' map = [ x^2 | x <- map ]
```

Listing 3. Working around `map`-detection

These implementations do not use the `map`-function, but use helper functions and arguments with this name in their (functionally correct) implementation. By using an identifier with the same name as a function in the outer scope, the outer reference is "shadowed" (replaced).

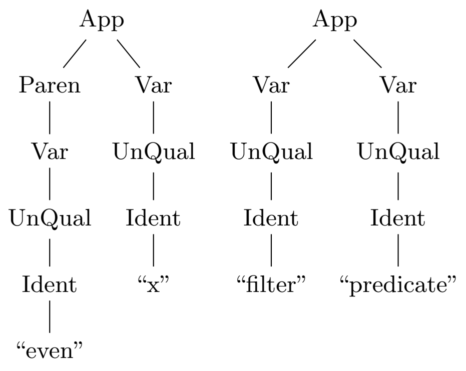
For such cases, it is necessary to implement proper scope management to detect what is being referenced: an argument, a self-defined function in an earlier scope or function from an imported module.

The approach Walker uses is similar to those found in many compilers [6]: while traversing the syntax tree, Walker builds a symbol table and opens a new scope when a function from a `where`-clause is entered. When referencing a function from an outer scope, the most recent scopes from the symbol table are removed until the depth is the same as the nesting depth of the called function.

The symbol table maps an identifier to the most recently encountered `Func` instance with that name. When an identifier name is an argument, that name is "blocked" in the symbol table: it is not a function reference the static analyzer can use, since its value is only known when executing the program.

In the example above, `square` is recognized not to use the built-in `map`, since it references the function in its `where`-clause. `square'` is recognized not to use `map`, because the name is shadowed by the function argument.

Figure 1. Syntax trees for right-hand sides of `predicate` and `filterTuples`



5.2 Helper functions and nested expressions

Using the symbol table, it is possible to follow the execution path, and verify the requirements based on the expressions in the helper functions that the student defined. Consider the following example, where a student should filter a list of tuples for even numbers in the first position (required is the use of the `even`-function inside `filterTuples`).

```
predicate (x, _) = (even) x
filterTuples = filter predicate
```

Listing 4. Filter using a helper function

The function `predicate` is used, but as an argument to another function. The requirement is fulfilled inside `predicate`, which is not the function that is currently being evaluated.

Walker uses the function references saved in the symbol table to evaluate the requirements on all functions which are (transitively) referenced from the main function. More specifically, it tries to pass a given requirement on any (possibly nested) expression found in any transitively used function that it not graded itself. Note: "using" in a functional language is not limited to having arguments applied. A function can also be used by passing it to a higher-order function.

For example, the right-hand sides of the functions in Listing 4 yield the syntax trees as found in Figure 1. Walker finds that this solution satisfies the requirement of using the predefined `even`-function in the following steps:

1. It (recursively) discovers all self-defined functions referenced from the main function using the symbol table. It finds that `predicate` is used in addition to `filterTuples`.
2. It removes the called functions that already have requirements associated to them, to prevent a "double punishment" when code from earlier assignments is reused. Both functions remain, since `predicate` is not graded itself.
3. All (possibly nested) expressions in the referenced functions are enumerated, along with the scope in which they appeared. In this example, those would be `filter predicate`, `filter` and `predicate`, combined with `(even) x`, `(even)`, `even` and `x`.
4. A reference to the imported function `even` is found: the requirement is satisfied.

The Haskell implementation for this algorithm uses the `Traversable` and `Typeable` classes (in conjunction with the `lens` library[8]) to efficiently explore the syntax tree, skipping the exploration of nodes that cannot contain expressions themselves.

The number of expressions found in the third step rapidly increases as the nesting of expressions grows deeper. As such, performance might suffer when assessing requirements on submissions with many (helper) functions and complicated nesting.

5.3 Recursion detection

A naive function discovery algorithm would not terminate when the student uses recursion, since each self-defined function would have a call to one or more self-defined functions, continuing forever.

Walker solves this problem by keeping track of the functions it has already seen. An option for this would be the construction of a call graph, like is done in most compilers [6] or in plagiarism detectors [9]. However, Walker uses a more basic approach: it stores a set of visited functions (`Func` instances), without any explicit links (such as graph edges) between them.

The reason for this is that, unlike those other tools, it is not necessary for the requirement validation to know via which path a function was referenced, but only that it *was* referenced.

Keeping track of a set of called functions also greatly reduces the complexity of checking if a student used recursion in their solution. Recursion detection of itself is not trivial, since it is not enough to check if a function references itself: it is also possible for multiple functions to form a recursive loop. Some examples of different forms of recursion are given in Listing 5.

```
rDirect x = x : rDirect x

rIndirect x = rDirect x

rWhere x = rWhereA x
    where
        rWhereA y = y : rWhere (y + 1)
        rWhereB z = z : rWhereA (z + 1)

rChain x = x : rChainB x
rChainB x = (x + 1) : rChainC x
rChainC x = (x + 2) : rChain (x + 3)
```

Listing 5. Different examples of recursion for an infinite list of increasing numbers

In order to detect recursion, the exploration of helper functions is used. Recursion is detected when a to-be-explored function was explored earlier, implied by the `Func` instance appearing in the set of earlier visited functions.

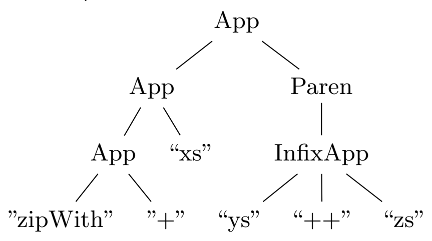
Since bindings in the `where`-clause are modelled as functions themselves, the recursion in `rWhere` is detected correctly as well.

5.4 Function application

When using specific functions, it can be desirable for teachers to require certain (kinds) of arguments to be passed to a specific function. A typical example would be to use a higher-order function with a lambda expression as an argument, which can be specified using the `argsExpr` field of `FuncUsage` (see Listing 2).

Retrieving the arguments of a function is not as trivial as it might seem, however. Consider the simplified syntax tree of the expression `zipWith (+) xs (ys ++ zs)`. Irrelevant intermediate nodes have been omitted.

Figure 2. Syntax tree for `zipWith (+) xs (ys ++ zs)` (simplified)



The function `zipWith` takes three arguments, but due to Haskell's partial application, it only has one sibling in the syntax tree. All other arguments are somewhere on higher levels of the tree, due to the left-associativity of function application.

When the algorithm described in Section 5.2 discovers the node containing `zipWith`, it must find the arguments to test the nested requirement on higher in the tree. This is not trivial using the *haskell-src-exts* library, since data structures in Haskell are generally not structured such that parents are accessible via the children.

Walker works around this problem by using a specific `Traversal` that only targets function application nodes (such as `App`, `InfixApp`, `RightSection` and others), allowing the syntax tree to be flattened into a list containing all arguments when used at the top-most level (such as the root in Figure 2). This eliminates the need to manually traverse until the deepest node containing the required function identifier.

5.5 Verifying inner requirements

Using the techniques described in the sections above (in particular Section 5.2), the remaining requirements found in Listing 2 can be verified relatively easily. Most are solved in the following way:

1. Enumerate all expressions referenced from the required function, including nested expressions and those found in helper functions.
2. Filter on the nodes required, for example list comprehension syntax or monads.
3. If necessary, transform the node into a new `Func` instance and recursively check the nested requirement on that instance.

When transforming into nested `Func` instances, values normally present (e.g., names and arguments) are often left empty. In other areas, the conversion does not faithfully represent the actual expression, but is constructed in a way that allows requirements checking in a `Func`-instance.

An example of this is the conversion of list comprehension, where generator statements (e.g., `x <- xs`) are treated as pattern bindings in a `where`-clause and qualifiers ("filters" in list comprehension) are converted similarly to pattern guards.

5.6 Type checking

There is one additional requirement that does not involve the literal traversal of the syntax tree. The required type signature of a function does not involve the search for language constructs in students' code, but does still require syntax tree operations.

The requirement is included in Walker to enforce the inclusion of hand-written type signatures in submissions. If code compiles, the type signature is correct, but the student might have taken these signatures from GHCi. Examples of such behavior would be the inclusion of `Functor` and `Foldable` class constraints by students who have not yet encountered these classes.

Type signature equality is not trivial to verify, however. Consider the different type signatures in the following code examples.

```
f :: a -> b -> c
f :: x -> y -> z

g :: (a, b) -> (,) a b
h :: [] a -> [a] -> a

j :: (Show a, Eq a) => a -> a
j :: (Eq a, Show a) => a -> a

k :: (Show b, Eq a) => ((,) b c -> c)
    -> IO ([] a)
k :: (Eq x, Show y) => ((y, z) -> z)
    -> IO [x]
```

Listing 6. Variations in type signatures

We observe the following mutations to the type signatures:

- Free type variables can be renamed (`f`).
- Special type constructor shorthands can be used (`g`/`h`).
- The ordering of type constraints can be mixed (`j`).
- Types can be parenthesized (`k`).
- Any of the mutations above can be mixed and nested (`k`).

The approach Walker uses is comparable to what *Ask-Elle* uses to compare entire functions: program transformations. Specifically, the following operations are (recursively) applied to both student and solution type signatures:

- Standardization of type variable names (also known as alpha-conversion).
- Expand shorthand type constructors.
- Sort type constraints by alphabetical order.
- Removal of parenthesis nodes from the syntax tree.

After these mutations, the syntax trees are compared for equality to verify the correctness of a type signature. The mutations ensure that type signatures that are essentially equal[2] are accepted, but that signatures using different types than intended are rejected.

6. VERIFICATION

In order to test the viability of the syntax-tree based approach, we tested Walker on different submissions by students of Functional Programming (mini)courses. This was done retroactively; as such, the students were not aware

[2]The additional information included by *haskell-src-exts* (such as line numbers) are excluded from this equality check.

of automated assessment when creating their code. Two different courses were used for testing, with different (assessment) properties: a relatively small introductory course, and a substantially larger project.

We created requirements for the final projects of these courses, and validated the results that Walker produced. We explicitly differentiate false-negatives (code wrongly rejected by Walker) from false-positives (code wrongly passing requirements of Walker).

During testing, no performance issues were encountered. Walker gave the results for each submission instantly on all hardware that the tests were performed on.

6.1 Small introductory course

The first course was for first-year students from IT-related studies, briefly introducing them to the fundamental concepts of Haskell and Functional Programming in general. As such, the requirements made for these exercises were relatively basic, use of recursion and re-use of their own functions being the primary targets.

We did not test the type signatures that some students included, for the simple reason that it was not explicitly required for the assignment. Consequently, most students did not include them, and most of those who did took them from GHCi's type inference. Only a handful out of 38 total submissions included proper type signatures.

We applied Walker to 38 different submissions, and manually checked the instances where a solution did not pass the requirements. Passing requirements were checked from a random sample of 10 submissions. In addition, grading was given as a pass or a fail, but documentation from the human assessor was not available.

From these 38 submissions, two were not syntactically correct due to indentation errors, and could thus not be processed by Walker. The other 29 submissions were assessed correctly: 10 of those contained one or more mistakes by students, the others were fully correct. The remaining grading reports for the remaining 7 submissions showed false-negatives due to renamed functions: students would often make spelling errors or abbreviate the requested function names. No other false-negatives occurred.

6.2 Large advanced course

The second course was intended for second-year CS students, finalized by a project involving parser combinators, instances of existing type classes and self-defined EDSL classes. Official grading criteria were mostly focused on re-use of existing code and avoidance of certain undesirable programming patterns.

6.2.1 Requirements creation

The human assessors used a spreadsheet with different grading criteria for evaluation of the submissions. These criteria consisted of some maximum number of points, with possible subtractions for often-seen (stylistic) mistakes and "other" errors, for which the assessor subtracted points manually where necessary.

Stylistic requirements for functions covered the in- or exclusion of certain functions and pattern matches, the correctness of particular type signatures and the usage of lambda expressions and Monads. All of these requirements could be converted to the `FuncReq` model, except for the type signatures using student-defined classes, since the name of the data class differed among students.

The remaining points were given for the compactness of the self-defined EDSL, which could not be modelled in Walker.

Table 1. Statistics for the second-year projects. 53 projects were assessed in total.

Phenomenon	Number of projects occurred in
Special style deductions (by human assessor)	14
Mistakes by human assessor	17
Deviating function names	6

6.2.2 Requirements verification

We applied Walker to 53 student submissions, comparing all outputs from Walker with the results from the different human assessors. As described above, it was not possible to fully grade the project using Walker alone: we only compared the results of the criteria that could be modelled in Walker.

During this test, we discovered a significant problem in the verification of type signatures. Type aliases break the equality of the syntax tree, while they do not really change the type: `String`, `[Char]` and `FilePath` all denote file paths, and result in `[Char]`. Walker could (in principle) solve this issue by normalizing using the alias definitions found in the syntax tree. However, such an approach would not work for imported aliases such as the examples above.

Besides this issue, Walker worked as expected. Significantly less naming errors, as seen for the smaller project, occurred. Walker also found a number of mistakes by the human graders, but the human graders also deducted points manually in some projects. The exact statistics can be found in Table 1. The mistakes by the human graders were corrected before the publication of the grades for this project.

6.2.3 Conclusion

While Walker was not able to fully grade all stylistic aspects of the project, it was able to accurately recognize all predefined grading criteria not involving type signatures and user-defined type classes. In hindsight, Walker could have been used to pre-populate those parts of the grading sheets, but human assessors would still be necessary to grade the user-defined classes and spot unconventional construct misuse.

7. DISCUSSION

The results in the preceding section showed very high accuracy for the criteria that were supported by Walker. While such accuracy is not entirely unexpected, it is important to discuss the possible shortcomings in our testing methodology, along with any unexpected results.

Most importantly, not all requirements that Walker can verify could be evaluated in different scenarios. The requirements least covered by our tests are the recursion detection and the validation of nested requirements. The recursion detection was tested on the smaller test group, but all students submitted code using direct self-recursion in a very small function. The validation of nested requirements (e.g., on the arguments of a specific function) was not tested in any of the projects, since such criteria were not used for any of the projects.

While more student-testing would have been preferable, extensive unit tests based on the examples given in this paper indicate that both recursion and requirements nesting work as intended. However, those examples were created by proficient Haskell programmers, rather than relatively inexperienced students. As such, some exceptionally strange code patterns might not be covered in the unit tests.

For the group of first-year students, all negative results from Walker were manually reviewed, but only a sample of the positive results were verified. The reason for this disparity is that it seemed us more likely to have false-negatives than false-positives due to the generally strict nature of pattern searching. Nevertheless, it would have been more desirable to verify all positive results as well. This was done on the projects by second-year students, however, and since those results showed no false-positives, we find it reasonable to assume those did not occur often in the first-year submissions as well.

Overall the higher accuracy is not surprising, given that the syntax tree-based approach was chosen specifically to allow more flexibility from student solutions (in comparison to program transformations) at the cost of being able to verify functional correctness. We verified this on two different projects and a combined total of 91 submissions. A larger and more diverse testing sample would help to further support this hypothesis, however.

Walker worked surprisingly fast, given the suspected performance issues as described in Section 5.2. One explanation is that Haskell's lazy evaluation prevents all expressions from being enumerated, since the algorithm will stop once a match has been found. The alternative is that these projects were simply not large enough for the problem to become significant. On the other hand: submissions by students are typically not very large, so it is unlikely that the problem would occur on other (types of) projects by students.

One final note is that, for the second-year students, Walker worked as expected, which does not imply that it found the same results as the human assessors. On some details, the interpretation of the grading criteria differed among the human graders. Walker behaved like any other grader, behaving consistently on all projects using a particular interpretation. For this reason, we did not list false-negative statistics: they would be heavily influenced by the subtle differences among all graders in general.

8. CONCLUSION

In this paper, we tested the viability of syntax tree-based assessment for automatic assessment of language constructs. We discussed different challenges that were recognized when using this approach, and a selection of techniques to overcome these issues. The most important issues were the need for a single function model, scope management and helper function exploration and expansion, in addition to variations of essentially equal type signatures. These techniques have been incorporated into the assessment prototype Walker.

By applying Walker to 91 submissions for two different projects, we found syntax tree-based assessment on language construct usage to be highly accurate, surpassing the accuracy of human graders when verifying style criteria such as the presence of lambda functions, recursion, pattern matches or calls to specific (imported) functions. The code transformations used for type signature verification were less effective, due to the possible incorporation of (user-defined) types or type aliases. Overall, we found syntax tree-based analysis to be highly useful for pre-filling most of the grading sheets, only requiring human assessors for custom classes and to catch exceptionally strange code patterns.

9. FUTURE WORK

The verification results showed numerous areas of improvement, to expand the number of grading criteria that can be automated with high accuracy.

One possible area of improvement is the comparison of type signatures. The current approach does not incorporate type aliases, which causes rejection of a signature using `String` instead of `FilePath`, for example. Since such aliases are not present in the students' source code, using source code from the Prelude would be an option. User-defined types do not need such a step, but the remaining obstacle is the resolution of type variables passed to the types. While solvable using a syntax tree-only approach, an alternative approach involving GHC might be beneficial as well.

Another interesting area is the automated assessment of self-defined data structures, as was done by humans in the projects of the second-year students. A possible direction for this would be to measure the similarity of the syntax trees of the student and the model solution. Such a solution should still allow for creative solutions, however.

Finally, the specification of requirements could possibly be an area of further research. An interesting alternative to the one presented in this paper is a requirement specified by a XPath query, allowing for more precise definition of where intermediate elements are allowed [11]. Another direction would be the for the grader to give requirement suggestions based on a model solution, allowing for generation of (for example) function usage or type signature requirements.

SOURCE CODE AVAILABILITY

The source code used for the prototype can be found at `https://github.com/VriesDeRick/haskell-checker`, licensed under the MIT license. We encourage interested educators or researchers to contact the author for possible questions or other inquiries.

ACKNOWLEDGEMENTS

The author would like to thank Marco Gerards for the helpful ideas, implementation suggestions and feedback on the prototype of Walker and this paper. Furthermore, he thanks the module coordinators from the University of Twente for granting access to the submissions to test on. Finally, the reviewers of the research track and the track chairs are thanked for their helpful comments and suggestions.

10. REFERENCES

[1] B. Canou, R. Di Cosmo, and G. Henry. Scaling up functional programming education: Under the hood of the OCaml MOOC. *Proc. ACM Program. Lang.*, 1(ICFP):4:1–4:25, Aug. 2017.

[2] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000.

[3] J. Hage, B. Vermeer, and G. Verburg. Research paper: Plagiarism detection for Haskell with Holmes. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research*, CSERC '13, pages 2:19–2:30, Open Univ., Heerlen, The Netherlands, The Netherlands, 2013. Open Universiteit, Heerlen.

[4] D. Heres and J. Hage. A quantitative comparison of program plagiarism detection tools. In *Proceedings of the 6th Computer Science Education Research Conference*, CSERC '17, pages 73–82, New York, NY, USA, 2017. ACM.

[5] J. Jeuring, L. T. van Binsbergen, A. Gerdes, and B. Heeren. Model solutions and properties for diagnosing student programs in Ask-Elle. In *Proceedings of the Computer Science Education Research Conference*, CSERC '14, pages 31–40, New York, NY, USA, 2014. ACM.

[6] L. T. Keith D. Cooper. *Engineering a compiler*. Elsevier/Morgan Kaufmann, second edition, 2012.

[7] H. Keuning, J. Jeuring, and B. Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Trans. Comput. Educ.*, 19(1):3:1–3:43, Sept. 2018.

[8] E. A. Kmett. *lens: Lenses, Folds and Traversals*. Available at `http://hackage.haskell.org/package/lens`. Accessed: 2019-06-22.

[9] M. L. Krammer. Plagiarism detection in Haskell programs using call graph matching. Master's thesis, Utrecht University, May 2011.

[10] G. R. Obaido. Structural analysis of source code plagiarism using graphs. Master's thesis, University of the Witwatersrand, Johannesburg, May 2017.

[11] J. Robie, M. Dyck, and J. Spiegel. XML Path Language (XPath) 3.1. Technical report, W3C, 2017.

[12] M. Striewe and M. Goedicke. A review of static analysis approaches for programming exercises. In M. Kalz and E. Ras, editors, *Computer Assisted Assessment. Research into E-Assessment*, pages 100–113, Cham, 2014. Springer International Publishing.