Teaching the Art of Functional Programming Using Automated Grading (Experience Report)

ALIYA HAMEER, McGill University, Canada
BRIGITTE PIENTKA, McGill University, Canada

Online programming platforms have immense potential to improve students' educational experience. They make programming more accessible, as no installation is required; and automatic grading facilities provide students with immediate feedback on their code, allowing them to to fix bugs and address errors in their understanding right away. However, these graders tend to focus heavily on the functional correctness of a solution, neglecting other aspects of students' code and thereby causing students to miss out on a significant amount of valuable feedback.

In this paper, we recount our experience in using the Learn-OCaml online programming platform to teach functional programming in a second-year university course on programming languages and paradigms. Moreover, we explore how to leverage Learn-OCaml's automated grading infrastructure to make it easy to write more expressive graders that give students feedback on properties of their code beyond simple input/output correctness, in order to effectively teach elements of functional programming style. In particular, we describe our extensions to the Learn-OCaml platform that evaluate students on test quality and code style.

By providing these tools and a suite of our own homework problems and associated graders, we aim to promote functional programming education, enhance students' educational experience, and make teaching and learning typed functional programming more accessible to instructors and students alike, in our community and beyond.

CCS Concepts: • **Social and professional topics → Computer science education**; **Student assessment**; • **Software and its engineering → Functional languages**;

ACM Reference Format:
Aliya Hameer and Brigitte Pientka. 2018. Teaching the Art of Functional Programming Using Automated Grading (Experience Report). *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 16 pages.

1 INTRODUCTION

Online programming environments, such as Ask-Elle [Gerdes et al. 2017] and the one used by the popular learning platform Codecademy [Sims and Bubinski 2011], offer immense potential to enhance students' educational experience in courses with high programming content. For one, having an entire development environment just a click away in a browser drastically lowers the entry barrier; students can start programming right away and concentrate on learning the course material, rather than getting frustrated and bogged down in the technical aspects of installing a new language and possibly learning a new editor. This is particularly the case for a lot of typed functional programming languages, such as OCaml, SML, or Haskell, where few integrated development environments (IDEs) exist, and support across platforms and operating systems differs widely. More significantly, online programming platforms feature autograding facilities that allow students to run and get feedback on their code while they are developing their programs, giving them the opportunity to fix bugs and address errors in their understanding right away. Having access to immediate feedback on their code has been recognized to significantly improve student learning outcomes and engagement with their assignments (see, e.g., [Gramoli et al. 2016; Sherman et al. 2013; Wilcox 2015]).

Authors' addresses: Aliya Hameer, School of Computer Science, McGill University, Canada, aliya.hameer@mail.mcgill.ca; Brigitte Pientka, School of Computer Science, McGill University, Canada, bpientka@cs.mcgill.ca.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

However, these automatic graders tend to focus heavily or even solely on the input/output correctness of an implementation, since this is the most straightforward property to assess without human interference. This approach neglects several other aspects of a student's code that are important to consider in order to give useful feedback; for instance, input/output testing cannot determine whether a student used the intended algorithm, language constructs such as pattern matching, or even good coding practices to solve a particular problem. As a consequence, students miss out on an entire class of valuable feedback that is extremely important for their learning.

In the Fall 2018 term, we piloted the use of the Learn-OCaml online programming platform [Canou et al. 2017, 2016] to deliver the course assignments in a second-year university course on programming languages and paradigms. The goal of the course is to provide a new perspective on fundamental concepts that are common in many programming languages through the lens of functional programming. It is taught primarily in OCaml and has had a combined enrollment of 642 students over the Fall 2018/Winter 2019 terms.

Learn-OCaml is an exercise environment originally developed for the massive open online course (MOOC) *Introduction to Functional Programming in OCaml* [Di Cosmo et al. 2015], colloquially known as "The OCaml MOOC". The platform allows students to write, typecheck, and run OCaml code directly in their browser. It includes an integrated grader: while working on their assignments, students can run the grader, see the grade they would get, and then improve their code until they are satisfied with the results. Graders are written in OCaml using the high-level grading combinators and lower-level grading primitives provided by Learn-OCaml's testing library. The bulk of the high-level portion of the library focuses on input/output testing, with much of the other functionality being as of yet underdeveloped; however, what makes the Learn-OCaml platform truly interesting, and why it offers so much promise for grading, is that grader code is given access to the student's code AST. This opens up a number of possibilities for performing static checks on the student's code using OCaml's compiler front-end libraries.

In this paper, we report on our experience in delivering and grading assignments using the Learn-OCaml platform in a classroom environment. Topics covered in our course and evaluated using Learn-OCaml include: higher-order functions; stateful vs. state-free computation; modelling objects and closures; using exceptions and continuations to defer control; and lazy programming. We discuss which concepts were easy to test using the platform and which were unexpectedly difficult; problems that arose when students started using the grader in ways other than what was intended; and the extensions to the platform that our experience motivated in order to better accomplish our teaching goals. The concrete contributions of this paper are the following:

(1) We describe how we used the built-in functionality of the Learn-OCaml grading library to evaluate students on a variety of functional programming concepts (Section 2).
(2) We release a set of homework problems and associated graders used in the Fall 2018 offering of the course, available upon request, for use by other instructors wishing to incorporate Learn-OCaml into their curriculum.
(3) In order to emphasize good software testing practices, even in the presence of an automated grader, we provide a mutation testing framework for Learn-OCaml. This framework can be used by instructors to require students to write unit tests for every function they implement, and evaluate these test cases on both correctness and coverage (Section 3).
(4) We present a set of extensions we have written for Learn-OCaml's grading library (Section 4). In particular, to teach students about good functional programming style, we release an extensible framework that supports style checking and provides suggestions (Section 4.2).[1]

[1]Our extensions to Learn-OCaml can be found at https://github.com/teaching-the-art-of-fp/learn-ocaml/tree/teaching-fp

In summary, we provide an account of our own experience and extend the out-of-the-box functionality of the Learn-OCaml platform in order to promote functional programming education, enhance students' educational experience, and make teaching and learning typed functional programming more accessible to both instructors and students.

2 LEARN-OCAML & OUR CURRICULUM

In this section, we describe our experience in using the Learn-OCaml platform for homework assignments in the Fall 2018 term. We discuss specific concepts in a programming languages course that were easy or difficult to evaluate using the automated grading library, and how switching to this mode of delivering assignments affected the course experience for students and teaching assistants in general. The lessons taken from this discussion will motivate the extensions to the platform that we describe in Sections 3-4.

2.1 Functional Programming Concepts

We used the Learn-OCaml platform to grade 9 homework assignments over the course of the Fall 2018 term. Here we highlight some of the main concepts these exercises tested and how we made use of the platform's grading library to evaluate students on these topics. After releasing some of these assignments it became clear when crucial aspects of the graders were overlooked, so we discuss also how these graders have been improved for future iterations of the course.

Datatypes and Pattern Matching. We introduced the students to user-defined datatypes by defining the types `suit` and `rank` as variant types, a playing `card` as a pair of a `rank` and a `suit`, and a `hand` as a list of `card`s as in Figure 1, inspired by an example in Harper [2013]. In the related exercise, students were asked to write functions to compare two cards by their rank, then by their suit and rank, and then to sort a hand of cards.

```
type suit = Clubs | Spades | Hearts | Diamonds
type rank = Six | Seven | Eight | Nine | Ten | Jack | Queen | King | Ace
type card = rank * suit
type hand = Empty | Hand of card * hand
```

Fig. 1. Datatype definition for a hand of playing cards

The built-in functionality of the Learn-OCaml platform worked quite well for this exercise. The grading library provided us with the student's code AST, and a simple syntactic check could determine whether the student used pattern matching to implement their functions. Input/output testing was then sufficient to verify that the student's code worked as intended. There was one issue we did not anticipate: several students tried to brute-force a function to compare two cards by writing a pattern matching with a case for each of the 81 possible pairs of ranks. This is easy to prevent by extending the syntactic check to confirm that the pattern matching contains a reasonable number of cases, and our grader has been modified to do this. To give an idea of the scope of this issue, we found that 63% percent of submissions used more than the 10 pattern matching cases that were needed for our solution.

Higher-Order Functions. Questions where we required students to use particular higher-order functions were relatively easy to test. A popular homework question (according to an end-of-term survey) was one where we defined a type `cupcake` as a pair of a price (float) and a list of ingredients (given as a variant type) as in Figure 2. Students were asked to write a function `allergy_free`, which takes as input a list of cupcakes and a list `allergens` of ingredients, and returns a list of only

```
type price = float
type ingredient = Nuts | Gluten | Soy | Dairy
type cupcake = Cupcake of price * ingredient list
```

Fig. 2. Datatype definition of a cupcake made up of possible allergens

those cupcakes that do not contain any of the given ingredients. They were required to implement this function using the higher-order functions `List.filter`, `List.for_all`, and `List.exists`. We were able to verify that an implementation met those requirements fairly easily with a simple syntactic check on the code AST.

Modelling Objects with Closures. One homework assignment asked students to write a function `new_account` which, given an initial password, would return a new value of type `bank_account` as defined in Figure 3. All operations on a bank account required that the correct password be given along with the operation, and providing the wrong password three times in a row was supposed to lock the account, not allowing any more operations to be performed. Problems involving modelling objects with state using closures are not difficult to grade automatically, just as it is not difficult to test implementations of simple classes in an object-oriented language.

```
type passwd = string
type bank_account = {
  update_passwd: passwd -> passwd -> unit;
  retrieve     : passwd -> int -> unit;
  deposit      : passwd -> int -> unit;
  balance      : passwd -> int
}
```

Fig. 3. Definition of a bank account "object"

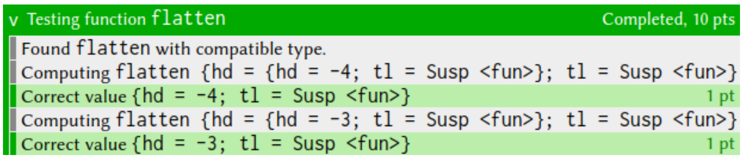
The unexpected difficulties in grading this assignment came from the fact that student code returned a bank account "object", which was a record of several *functions*. The high-level utilities in Learn-OCaml's grading library focus on comparing the output of running the student's code to some expected output; in this case the outputs were "objects" which we wished to run our own series of tests on, instead of simple values that we would want to compare to some others. Testing the outputs in this way while providing the students with informative feedback required us to copy a lot of code from the grading library and modify it slightly for our purposes. This was a problem that came up repeatedly in other contexts and is not specific to the topic of modelling objects.

Control Flow. Students implemented a parser for a simple arithmetic language twice: once using exceptions for control flow, and once using continuations. Both were very easy to test using the built-in functionality of the grading library. Since the testing functions by default compare exceptions raised by the student's code and the solution as part of input/output testing, the only extra work we had to do in this case was adjust how exceptions carrying more complex values like lists were printed, so that the feedback given to the students was properly meaningful. In the case of the implementation using continuations, testing the student's code with a few predefined continuations with different output types was sufficient to confirm that they were indeed calling the given continuation with the expected inputs.

Lazy Programming. One of the final assignments of the term focused on lazy lists, as defined in Figure 4. Delayed computation is modelled by a function that takes an input of type `unit`; lazy

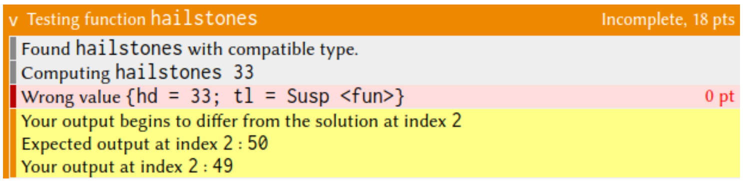
```
type 'a susp = Susp of (unit -> 'a)
let force (Susp s) = s ()
type 'a lazy_list = {
  hd: 'a;
  tl: ('a lazy_list option) susp
}
```

Fig. 4. Definitions for lazy computation



```
v Testing function flatten                                         Completed, 10 pts
Found flatten with compatible type.
Computing flatten {hd = {hd = -4; tl = Susp <fun>}; tl = Susp <fun>}
Correct value {hd = -4; tl = Susp <fun>}                                       1 pt
Computing flatten {hd = {hd = -3; tl = Susp <fun>}; tl = Susp <fun>}
Correct value {hd = -3; tl = Susp <fun>}                                       1 pt
```

(a) Default printed representation of a lazy list



```
v Testing function hailstones                                     Incomplete, 18 pts
Found hailstones with compatible type.
Computing hailstones 33
Wrong value {hd = 33; tl = Susp <fun>}                                         0 pt
Your output begins to differ from the solution at index 2
Expected output at index 2 : 50
Your output at index 2 : 49
```

(b) Output of our custom lazy list comparator

Fig. 5. Grading reports for functions involving lazy lists

lists are defined as a record of a head element and the delayed computation for the tail of the list. Students were asked to implement `map`, `append`, and `flatten` functions for lazy lists, and then to use these to write a function `permute` which lazily computes all the permutations of a given (non-lazy) list.

Students were also tasked with implementing a function `hailstones` that lazily generates the hailstone sequence (also known as the Collatz sequence) starting from a given positive integer. The hailstone sequence $(a_i)_{i\in\mathbb{N}}$ starting from a positive integer n is defined as follows:

$$a_0 = n$$
$$a_{i+1} = \begin{cases} \frac{a_i}{2} & a_i \text{ is even} \\ 3a_i + 1 & a_i \text{ is odd} \end{cases}$$

For up to extremely large values of n, the hailstone sequence starting from n has been checked to always eventually reach the cycle $4, 2, 1, 4, 2, 1, \cdots$. However, while it is conjectured that this is true for all positive integers n, this has not been proven; thus students were asked to generate the sequences using lazy lists.

Automatically grading these functions was predictably quite difficult, since potentially infinite structures were involved. One difficulty was in simply giving the students meaningful feedback. Lazy lists, by their definition, do not have an easy-to-read printed representation, and since they have the potential to be infinite, naïvely trying to print out the elements of a lazy list is not practical. The default printed representation of a lazy list is shown in Figure 5a, and is clearly not useful to students trying to replicate a failed test case. An acceptable solution might be to print out the elements of the list only up to a certain bound.

Comparing a lazy list against an expected output could not be done with 100% certainty, as the structures involved could be infinite. Indeed, in the case of the `hailstones` function, all outputs were expected to represent infinite sequences. We wrote a custom tester that steps through two lazy lists to compare them pairwise until a certain value is reached (in the case of `hailstones`, that value was `1`). An example error report given by this tester is shown in Figure 5b.

Other Topics. Other concepts covered in the assignments over the term include references and state, and implementation of a type-checker and evaluator for a small toy language. These topics did not bring any new insights to the table in terms of implementing graders, so we do not give further detail on them here.

2.2 General Lessons

Apart from the lessons learned about writing effective graders for a variety of important concepts in a programming languages course, we found several takeaways about writing and using automated graders in general. Many of these have been previously encountered by instructors introducing automated grading into their courses, but are not well-documented in the functional programming community. We summarize a few key difficulties here.

Writing Graders is Time-Consuming. We released assignments on the Learn-OCaml platform weekly, and the graders were written by the teaching assistants during the term. It became clear very early on that this was not a sustainable practice. We would like to emphasize that letting the selection of high-level functions provided by the grading library prescribe the criteria for the graders, and simply writing grader code accordingly, is not sufficient to write a good grader. Writing an effective grader requires:

(1) Deciding on a sufficient set of test cases and how to weight them;
(2) Determining what syntactic checks should be performed on the student's code;
(3) Anticipating unusual solutions that students may come up with to the homework problems, and making sure that the planned grading procedures will handle them properly;
(4) Actually implementing the grader;
(5) Thoroughly testing the grader.

Notably, implementing the graders using the libraries provided by Learn-OCaml is only a small part of the process. We have also omitted a step (0) of deciding if an assignment is even suited to automated grading at all. An example of an assignment that may not be suitable would be one that requires students to define their own datatype and then write functions that take values of this type as input or return them as output. Since the datatype is declared in the student's code, we cannot generate values of that type in the grader, nor do we have anything to compare values of that type to if they are returned as output. It could be possible to test this code by requiring students to do the extra work of writing conversion functions between the new datatype and some type built in to OCaml, but particularly for earlier assignments we would prefer not to do this.

It is also infeasible to keep coming up with graders for new assignment problems year after year. Therefore, we plan to build a repository of pre-written graders, starting with the ones we have written for the Fall 2018 term, and re-use them over the years. As a contribution of this paper, we release these homework problems and associated graders to other instructors teaching courses in OCaml upon request. We hope for this to be a starting point for a bank of tried and tested homework assignments that can be shared among instructors in the community at large, making adoption of the Learn-OCaml platform more accessible to those who wish to make the switch.

Improper Use of the Grader. A major issue we experienced over the course of the term was that students were becoming overreliant on the automated grader and using it as a substitute for testing their code on their own. The Learn-OCaml platform provides an interactive toplevel for students to test their code; however, in an end-of-term survey, 35% of students responded that they used the toplevel to test their code “rarely” or “never”. On earlier assignments, students could get away with writing their programs by trial and error based on the grader’s feedback. Later assignments where the questions were more complicated made this strategy much more difficult and prohibitively time-consuming. A common complaint through the term was that the grader did not give the expected output for failed test cases, and students were thus required to figure out the expected outputs themselves. This phenomenon has been previously observed when relying on automated grading for programming assignments; for some examples, see [Chen 2004; Edwards 2003].

One possible solution to this problem is to limit the number of times students can run the grader on their submissions, thus requiring them to test their code carefully before using up one of their limited tries. This has been observed by Pettit et al. [2015] to increase the average quality of student submissions.

We have decided to take a different route, in which we require students to write their own tests and make use of the automated testing infrastructure to grade students on their test quality. A similar approach has been tried in a third-year computer organization course by Chen [2004], and is still in use in the course today. We give details on this solution, and how we have implemented it as a library for Learn-OCaml, in Section 3.

Limitations of the Grading Library. The grading library that Learn-OCaml provides specializes in testing input/output correctness against a solution and in performing simple syntactic checks on the student’s code: for instance, checking that a certain name occurs syntactically in the body of a function, or verifying that a function is implemented without using `for` loops. More complicated syntactic checks one might like to perform, such as checking if a certain call is a tail call, are doable with the library’s provided functionality, but require users to write a lot of extra code from scratch.

A major syntactic property of a student’s code that we wish to evaluate is *code style*. The meaning of code style is subjective among instructors, but it is commonly taken into account in some form when grading programming assignments manually; when using Learn-OCaml’s provided library out of the box, students miss out on this entire class of valuable feedback. Again, writing style checks is possible with the current grader library, but involves a lot of extra code.

We have written extensions for Learn-OCaml’s grading library to mitigate this issue, which we discuss in Section 4.

3 ENCOURAGING TEST-DRIVEN DEVELOPMENT

A drawback of having a readily-accessible automatic grader for the homework assignments is that it is tempting for students to rely on the grader to tell them if their code is correct, instead of testing their own code thoroughly. We have implemented an extension to the Learn-OCaml platform similar to the idea described in Chen [2004], by which we can leverage the automated grader to evaluate students not only on their code, but also on unit tests that they write for it.

We assess the quality of a student’s tests using a strategy based on *mutation testing* [DeMillo et al. 1978]. Suppose that we wish the student to write tests for a function `pow` such that, for natural numbers `n` and `k`, `pow n k` evaluates to n^k (we do not care what `pow 0 0` returns). The student provides the tests as a list of input/expected output pairs as shown in Figure 6a. Programmed into the grader is a set of buggy implementations of the `pow` function, which we call *mutants*. An example of some mutants of the `pow` function is given in Figure 6b: one buggy implementation simply multiplies its inputs instead of performing exponentiation, and the other has a typo in the first line of the `if`-expression, resulting in all calls returning `0`.

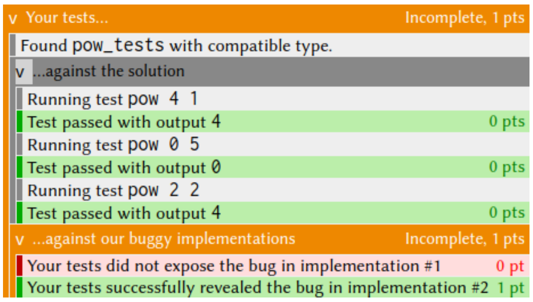
```
let pow_tests = [
  ((4, 1), 4);
  ((0, 5), 0);
  ((2, 2), 4)
]
```

(a) A set of sample unit tests

```
let pow n k = n * k

let pow n k =
  if k = 0 then 0
  else n * pow n (k - 1)
```

(b) A set of sample mutants



```
v Your tests...                                              Incomplete, 1 pts
Found pow_tests with compatible type.
v ...against the solution
Running test pow 4 1
Test passed with output 4                                              0 pts
Running test pow 0 5
Test passed with output 0                                              0 pts
Running test pow 2 2
Test passed with output 4                                              0 pts
v ...against our buggy implementations                      Incomplete, 1 pts
Your tests did not expose the bug in implementation #1                 0 pt
Your tests successfully revealed the bug in implementation #2          1 pt
```

(c) Sample grading report for mutation testing

Fig. 6. Mutation testing extension for Learn-OCaml

The student's tests are first checked for correctness by running them against the correct solution code, as seen in the first section of the report in Figure 6c. Including this functionality allows students to experiment with how they think the functions they are asked to implement should work, by writing test cases and then verifying their understanding against the solution.

Once all of the student's tests have been deemed correct, they are next checked for coverage by running them against each of the mutants. Given a specific mutant, if at least one of the student's test cases results in an output from the mutant that does not match the indicated expected output, the student's tests are said to have exposed the bug in that implementation. Otherwise the tests are insufficient to reveal the bug, and the student must revisit their test cases and consider what cases they might not be covering. In Figure 6, the given test cases are enough to uncover the bug in the second mutant in 6b, but not the first, as the test cases are not enough to distinguish exponentiation from simple multiplication. Adding the test case `((2, 3), 8)` to `pow_tests` in Figure 6a would be sufficient for the test suite to uncover the bugs in both mutants.

Normally in a software development setting, a large number of mutants are generated automatically by making small modifications to the code being tested, such as replacing a mathematical operator with a different one, or replacing a variable name with a constant or a different variable; this is meant to mimic common programmer errors. A drawback of this is that since so many mutants need to be generated and then executed against the test suite for this to be effective, the computational cost of mutation testing can be quite high [Jia and Harman 2011]. In our use case, we already know what a desired solution looks like. We can thus write a small number of very specific mutants manually while writing the grader, and use these for mutation testing. Writing the mutants manually allows us to isolate certain cases that we want to make sure students are testing for. Mutation testing is thus ideal here since we can achieve a high level of effectiveness with a low computational cost.

Having students write their unit tests as input/output pairs is simple enough that they can start writing tests right away on their very first homework assignment, given a proper template. However, it does not allow them to account for console output, express that a test case should raise an exception, or write test cases for functions involving randomness, for example. The simple input/output testing functionality is sufficient for the non-error cases in the vast majority of functions we have students write, so we believe this is not a significant drawback, and we expect based on the experience by Chen [2004] that it will help us significantly in achieving our broader learning goal of emphasizing good software development practices for our students.

4 WRITING MORE POWERFUL GRADERS MORE EASILY

As it is, the grading library provided by the Learn-OCaml platform has a lot of potential for writing powerful graders, as the library allows graders to access the student's code AST. However, the library only provides helpers for carrying out simple syntactic checks on the parse tree. In order to take full advantage of having access to the AST, users are required to write a lot of extra code, and this code needs to be copied and pasted into each grader separately. We have also come across situations where we would like to have access to the full typed AST: for instance to make use of scoping information, or because we might want to look at types to determine whether a student is using higher-order functions without knowing which specific functions we are looking for. Over the course of our first term using the platform, we wrote several extensions to the grading library aimed at increasing the out-of-the-box functionality of Learn-OCaml's syntax checking, so that future users of the platform can more easily write powerful and expressive graders.

4.1 Checking Tail Recursion

Tail recursion is one of the first topics we cover in our course; multiple questions on the first couple of exercises specifically ask students to implement certain functions in a tail-recursive manner. Thus one of the first issues we encountered when using the platform was: How can we check that a student's implementation is actually tail recursive?

The first, dynamic, approach that we used was simply to call their function with an extremely large input; if their function was not tail recursive, this would result in a stack overflow which the platform would detect. This seemed to work well enough for a while, although it would have been desirable to have a way of pointing out which recursive call was not a tail call.

However, we ran into a problem in the Winter 2019 term when we gave an example using exponentiation by repeated squaring, as implemented in Figure 7, and asked students to implement a tail-recursive version of this algorithm. By design, exponentiation by repeated squaring uses stack space logarithmic in k, so that giving even the largest possible integer value for k as argument to the non-tail-recursive implementation in the figure will not result in a stack overflow. This means that we cannot try to verify that a student's implementation is indeed tail recursive using a dynamic approach.

```
let rec fast_pow n k =
  if k = 0 then 1
  else if k mod 2 = 1 then
    n * fast_pow n (k - 1)
  else
    let x = fast_pow n (k / 2) in
    x * x
```

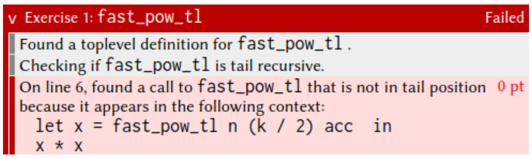
Fig. 7. Exponentiation by repeated squaring

The OCaml compiler features a `[@tailcall]` annotation which can be added to recursive calls so that the compiler will emit a warning if they are not tail calls; however, in the grader we do not have access to compiler warnings, and we would prefer students not to need to annotate all of their recursive calls.

We have thus written our own syntactic check which attempts to determine if a function is tail recursive and incorporated it as part of a library in Learn-OCaml. The problem of determining whether or not a function is tail recursive is undecideable in the general case due to factors like aliasing and mutation, and we do not attempt to address such factors in our implementation; however, we feel that our implementation is sufficient for handling the scenarios that come up in practice with student code. We traverse the code AST of the function in question and check if we can find any recursive call that is not a tail call. If so, we flag it as an error and provide the student with the context in which the function call appears outside of tail position. As an example, consider an attempt to implement exponentiation by repeated squaring tail-recursively using the helper function `fast_pow_tl` given in Figure 8a. This function is not tail recursive, since a recursive call to `fast_pow_tl` appears in a binding of a `let`-expression on line 6. Our syntactic check generates a report as shown in Figure 8b, informing the student that a recursive call has been found outside of tail position and pinpointing the location where this happens.

```
1  let rec fast_pow_tl n k acc =
2    if k = 0 then acc
3    else if k mod 2 = 1 then
4      fast_pow_tl n (k - 1) (n * acc)
5    else
6      let x = fast_pow_tl n (k / 2) acc in
7      x * x
```

(a) Attempt to implement exponentiation by repeated squaring tail-recursively


v Exercise 1: `fast_pow_tl` Failed

Found a toplevel definition for `fast_pow_tl` .

Checking if `fast_pow_tl` is tail recursive.

On line 6, found a call to `fast_pow_tl` that is not in tail position 0 pt
because it appears in the following context:

```
  let x = fast_pow_tl n (k / 2) acc  in
  x * x
```

(b) Grading report generated for the implementation in (a)

Fig. 8. Tail recursion checking in Learn-OCaml

4.2 Style Checking

One major class of feedback students miss out on when their code is graded automatically is feedback on their *code style*. What constitutes good style is a subject of debate: it can concern identifier names, use of whitespace, comments, number of characters in a line, etc. We are not interested in syntactic concerns such as these; following Michaelson [1996], we instead focus on teaching students good style on a *semantic* level. By "semantics" here we refer to the intended purpose of specific language constructs. Good style on a semantic level can thus be described as "making appropriate use of language constructs": for instance, in a typed functional programming setting this includes using pattern matching instead of explicit selectors when appropriate; making use of partial application instead of defining eta-expanded functions as inputs to higher-order functions such as `List.map`; and not making comparisons with boolean values. It is our experience that semantic style errors that students make are often rooted in conceptual misunderstandings about the constructs in question.

As an example, consider the following function, which checks if an integer `n` is odd:

```
let odd n =
  if n mod 2 = 1 then true
  else false
```

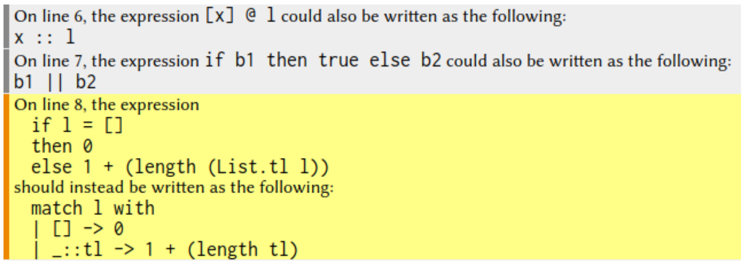
```
[e1] @ e2         ⟹     e1 :: e2

if e1
then true         ⟹     e1 || e2
else e2

if e = []               match e with
then e1           ⟹     | [] -> e1
else e2                 | hd :: tl -> e2′
```

where e2 = [`List.hd e` / hd, `List.tl e` / tl] e2′

(a) Sample of the rewrite rules implemented in the style checker extension



On line 6, the expression `[x] @ l` could also be written as the following:

```
x :: l
```

On line 7, the expression `if b1 then true else b2` could also be written as the following:

```
b1 || b2
```

On line 8, the expression

```
  if l = []
  then 0
  else 1 + (length (List.tl l))
```

should instead be written as the following:

```
  match l with
  | [] -> 0
  | _::tl -> 1 + (length tl)
```

(b) Sample feedback report corresponding to code that triggers the rewrite rules in (a)

Fig. 9. Style checking extension for the Learn-OCaml platform

This could be written more elegantly as:

```
let odd n =
  n mod 2 = 1
```

However, students will sometimes write the former due to confusion about how conditional expressions relate to boolean values. Even after being given a general warning never to write expressions of the form **if** ≪boolean-expression≫ **then true else false** some students will continue to do so. We believe that this is due to students not connecting this general form to specific instances of it that they write in their code. It is this observation that led us to the design of our style-checking extension to the Learn-OCaml platform.

Inspired by Michaelson [1996]'s work on a style checker for SML, we have implemented a style checker as a program that traverses the student's code AST and suggests the application of various rewrite rules in order to improve the style of the student's code. Figure 9a gives a small sample of the formal rewrite rules we have implemented. In the first example, a list element `e1` is put into a singleton list so that the list `e2` can be appended to it, instead of simply using the `::` ("cons") operator which has the same purpose. The second example shows a conditional statement that could be transformed into a use of the `||` ("or") operator. For the final example, we emphasize that pattern matching on list structure is strictly preferable to comparing to the empty list (`[]`) and using the explicit list selectors `List.hd` and `List.tl`. A full list of the rewrite rules we have implemented can be found in the Appendix. We have also included checks that will emit a warning if the student's code uses more than a given number of cases in any one pattern matching or conditional expression.

When the style checking extension is enabled, a section on code style will be added to the student's feedback report whenever they run the autograder on their code. A sample such report can be found in Figure 9b. Note that we have two categories of style feedback: *suggestions* (gray background) and *warnings* (yellow background). Suggestions are those rewritings that one may or may not want to apply depending on concerns such as code readability. Warnings are those rewrites that we feel should always be applied. Using features already present in Learn-OCaml's grading library, it is possible to have the style report contribute points to the student's grade, or to refuse to grade the functionality of the student's code if they have more than a certain number of warnings.

We believe that by continuously emphasizing ideas of good style to our students with examples involving their own code, this style checking extension will be a great help in teaching our students elements of good functional programming style.

4.3 Implementation of Grader Extensions

The extensions to the grading libraries discussed in this section are implemented atop our own augmented version of the OCaml parse tree, which we call `Typed_ast` to differentiate from the OCaml compiler's internal `Typedtree` structure. By design, grader code is only run on code that typechecks; therefore, we can make use of OCaml's compiler front-end to build up the `Typedtree` structure from the student's code AST and get access to typing and scoping information for all identifiers. However, this internal `Typedtree` structure is difficult to work with, and the process of going from the untyped AST to a `Typedtree` is non-invertible, which is an issue when we wish to print out rewritten code back to the student as in Figure 9. Thus, we build our own `Typed_ast` by extracting the information we need from the `Typedtree` and using it to annotate the untyped AST. We annotate each expression in the untyped AST with its type and the typing environment in which it was type-checked, and each identifier with scoping information, which includes its full resolved module path and a unique identifier so that shadowed identifiers can be distinguished.

Basing all of our syntactic checks on these `Typed_ast`s instead of the untyped AST means that our implementations are scope-safe:

- Tail call checking continues to work as expected when the name of the function in question has been shadowed inside the body of the function.
- In the style checker, AST transformations that introduce references to predefined identifiers (such as `List.tl`, `&&`, or `not`) are not suggested in contexts where the identifier to be introduced has been overshadowed by another definition.

5 RELATED WORK

Using automated grading for programming classes has been suggested as early as 1960 [Hollingsworth 1960] and the practice has become common in higher education, with the development of systems such as ASSYST [Jackson and Usher 1997], Ceilidh [Benford et al. 1995], Web-CAT [Edwards and Perez-Quinones 2008], BOSS [Joy et al. 2005], and many more. Ala-Mutka [2005] provides a survey of the field up to the mid-2000s. These systems generally follow a simple model: students write code on their own machines and submit their work to a server. The server then executes the student's code in some sort of sandboxed environment and, usually after some delay, sends some report back to the student.

In our experience, the Learn-OCaml online programming platform provides a more expressive and flexible environment. Students do not need to install anything, and can work on and grade their code all in one place. Instructors can not only take advantage of the simple out-of-the-box support for checking input/output correctness of a students' program, but also tap into its potential to teach software testing practices via mutation testing and suggestions on elements of programming style. We believe this can substantially enhance the learning experience of the student.

The DrRacket IDE, augmented with the Handin Server plugin[2], provides a similar experience for students as the Learn-OCaml platform. Students download and install DrRacket and a course-specific plugin and are then able to submit their work simply by clicking a "Handin" button in the IDE, which uploads their work to a server, runs some instructor-defined tests, and returns the results of these tests to the student. The test automation framework allows instructors to test for input/output correctness, perform syntactic checks on the student's code AST, and evaluate the code coverage of student test suites. Student submissions remain on the server after tests are run for teaching staff to carry out further manual grading if desired. Our work on the Learn-OCaml platform is in a similar spirit to the DrRacket project: the DrRacket IDE has made languages in the Scheme family popular in the university setting, but there exists no such platform for typed functional programming languages in the ML family.

Other tools exist that suggest code refactorings for better style in the sense of our style checker. Examples of these include Tidier, for Erlang [Sagonas and Avgerinos 2009], and HLint[3], for Haskell. Tidier functions simlarly to our style checker, suggesting a variety of program transformations based on ideas such as using appropriate language constructs and making good use of pattern matching. HLint focuses heavily on pointing out functional equivalences, such as replacing `concat (map f x)` with `concatMap f x`, though it includes rules for several of the transformations we have implemented for our style checker as well. The set of checks can be extended with custom rules added via a configuration file.

6 CONCLUSION

In this paper we have reported on our experience in using the Learn-OCaml platform, which was originally developed as an exercise environment for a MOOC, in a university setting. We have discussed how we used the features of the platform to test students on a variety of concepts that come up in a programming languages course, the problems that came up while doing so, and the extensions to the platform that we subsequently implemented for evaluating students on properties of their code beyond input/output correctness.

There still remains work to be done. In the future, we plan to work on several fronts to incorporate new ideas into Learn-OCaml:

(1) *Plagiarism detection.* Systems for measuring code similarity such as MOSS[4] [Schleimer et al. 2003] are commonly used in large programming classes to detect possible cases of plagiarism by comparing student code submissions. No such mechanism currently exists in the Learn-OCaml platform.
(2) *Interactive lessons.* In addition to the exercise functionality, the Learn-OCaml platform allows instructors to create interactive tutorials and lessons for their students. In the future we would like to convert our course notes to this format to help facilitate independent learning.
(3) *Typed holes.* We encourage our students to develop their programs incrementally by "following the types"; as such, it would be quite useful to be able to type-check partial programs, as in the live programming environment Hazel [Omar et al. 2017]. Being able to insert typed holes into incomplete programs would allow students to check, for example, the type of an argument that they need to provide to a function, or the type that a certain sub-expression needs to evaluate to in order for the full expression to have a desired type.

[2]https://docs.racket-lang.org/handin-server/Handin-Server_and_Client.html
[3]https://github.com/ndmitchell/hlint
[4]https://theory.stanford.edu/ aiken/moss/

(4) *Incremental stepping*. The DrRacket environment includes an algebraic stepper[5] which allows students to step through their code and visualize what conceptionally happens when evaluating a given program. Cong and Asai [2016] presents an implementation of a stepper for a small subset of OCaml, implemented in a fork of the Caml language.

(5) *Complexity analysis*. Aside from correctness, test quality, and code style, another metric of student code that would be useful to assess automatically is time complexity. Hoffmann et al.'s work on automatically analyzing resource bounds of OCaml programs has led to the development of Resource Aware ML (RAML) [Hoffmann et al. 2012, 2017], a resource-aware version of the OCaml language. Currently only a subset of OCaml is supported and the language is not ready for deployment in a learning setting. In the future it would be interesting to see if these ideas could be ported to Learn-OCaml.

By sharing our experience and our tools together with a suite of homework problems, we hope to make it easier for other instructors to offer high-quality functional programming courses and allow us to promote best functional programming practices in our community and beyond.

ACKNOWLEDGMENTS

This material is based upon work supported by the Natural Science and Engineering Council (NSERC) under Grant No. 206263 and by the OCaml Software Foundation under Grant No. 249299. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of NSERC or the OCaml Software Foundation.

REFERENCES

Alex Aiken. [n. d.]. MOSS: A System for Detecting Software Similarity. ([n. d.]). https://theory.stanford.edu/~aiken/moss/

Kirsti M Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* 15, 2 (2005), 83–102. https://doi.org/10.1080/08993400500150747

Steve D Benford, Edmund K Burke, Eric Foxley, and Christopher A Higgins. 1995. The Ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the 33rd annual on Southeast regional conference*. ACM, 176–182.

Benjamin Canou, Roberto Di Cosmo, and Grégoire Henry. 2017. Scaling up functional programming education: under the hood of the OCaml MOOC. *Proceedings of the ACM on Programming Languages* 1, ICFP (aug 2017), 1–25. https://doi.org/10.1145/3110248

Benjamin Canou, Grégoire Henry, Çagdas Bozman, and Fabrice Le Fessant. 2016. Learn OCaml, An Online Learning Center for OCaml. In *OCaml Users and Developers Workshop 2016*.

Peter M Chen. 2004. An Automated Feedback System for Computer Organization Projects. *IEEE Transactions on Education* 47, 2 (May 2004), 232–240. https://doi.org/10.1109/te.2004.825220

Youyou Cong and Kenichi Asai. 2016. Implementing a stepper using delimited continuations. *contract* 1 (2016), r1.

Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.

Roberto Di Cosmo, Yann Regis-Gianas, and Ralf Treinen. 2015. Introduction to Functional Programming in OCaml. (October 2015). https://www.fun-mooc.fr/courses/parisdiderot/56002/session01/about

Stephen H Edwards. 2003. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *International Conference on Education and Information Systems: Technologies and Applications (EISTA'03)*, Vol. 3.

Stephen H Edwards and Manuel A Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 328–328.

Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. 2017. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 65–100.

Vincent Gramoli, Michael Charleston, Bryn Jeffries, Irena Koprinska, Martin McGrane, Alex Radu, Anastasios Viglas, and Kalina Yacef. 2016. Mining Autograding Data in Computer Science Education. In *Proceedings of the Australasian Computer*

[5] https://docs.racket-lang.org/stepper/index.html

Science Week Multiconference (ACSW '16). ACM, New York, NY, USA, Article 1, 10 pages. https://doi.org/10.1145/2843043.2843070

Robert W. Harper. 2013. *Programming in Standard ML*. (draft available at https://www.cs.cmu.edu/~rwh/isml/book.pdf).

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. *Lecture Notes in Computer Science* (2012), 781âĂŞ786. https://doi.org/10.1007/978-3-642-31424-7_64

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. *ACM SIGPLAN Notices* 52, 1 (Jan 2017), 359âĂŞ373. https://doi.org/10.1145/3093333.3009842

Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3, 10 (1960), 528–529.

David Jackson and Michelle Usher. 1997. Grading Student Programs Using Assyst. *ACM SIGCSE Bulletin* 29, 1 (1997), 335–339. https://doi.org/10.1145/268085.268210

Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sep. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The boss online submission and assessment system. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 2.

Greg Michaelson. 1996. Automatic analysis of functional program style. In *Australian Software Engineering Conference*. 38–46. https://doi.org/10.1109/aswec.1996.534121

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. *ACM SIGPLAN Notices* 52, 1 (2017), 86–99.

Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An Empirical Study of Iterative Improvement in Programming Assignments. In *46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, 410–415. https://doi.org/10.1145/2676723.2677279

Konstantinos Sagonas and Thanassis Avgerinos. 2009. Automatic refactoring of Erlang programs. *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming - PPDP âĂŹ09* (2009). https://doi.org/10.1145/1599410.1599414

Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing. *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD âĂŹ03* (2003). https://doi.org/10.1145/872757.872770

Mark Sherman, Sarita Bassil, Derrell Lipman, Nat Tuck, and Fred Martin. 2013. Impact of Auto-grading on an Introductory Computing Course. *J. Comput. Sci. Coll.* 28, 6 (June 2013), 69–75. http://dl.acm.org/citation.cfm?id=2460156.2460171

Zach Sims and Ryan Bubinski. 2011. Codecademy. (2011). http://www.codecademy.com

Chris Wilcox. 2015. The role of automation in undergraduate computer science education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 90–95.

A STYLE CHECKER REWRITE RULES

Below we give a full list of the rewrite rules for the style checker described in section 4.2.

```
                   e = true   ⟹   e

                  e = false   ⟹   not e

    if e then true else false   ⟹   e

    if e then false else true   ⟹   not e

     if e1 then true else e2   ⟹   e1 || e2

    if e1 then e2 else false   ⟹   e1 && e2

              match e1 with        let pat = e1 in
              | pat -> e2     ⟹   e2

                  [e1] @ e2   ⟹   e1 :: e2

                     e @ []   ⟹   e

                     [] @ e   ⟹   e

                  if e = []        match e with
                  then e1     ⟹   | [] -> e1
                  else e2          | hd :: tl -> e2'*

        fun pats1 pats2 ->    ⟹   fun pats1 -> e**
          e pats2
```

* where `e2` = [`List.hd e` / `hd`, `List.tl e` / `tl`] `e2'`

** where no pattern variable from `pats2` appears free in `e`