Teaching the Art of Functional Programming Using Automated Grading

Aliya Hameer

School of Computer Science

McGill University, Montreal

March 2020

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science.

Abstract

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 thesis, 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 present our extensions to the Learn-OCaml platform that evaluate students on test quality and code style. We then describe how we used these extensions in a later offering of our course to better pursue our teaching goals, and provide data from a study on students' code submissions to determine how students were using the platform.

By providing our lessons learned over these semesters of using Learn-OCaml, as well as our new 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 all across the globe.

Abrégé

Les plateformes de programmation en ligne ont un potentiel immense d'améliorer l'expérience éducationnelle des étudiants. Elles offrent une expérience de programmation plus accessible puisqu'elles ne requièrent pas d'installation de configuration de la part des étudiants. De plus, les systèmes de notation automatique permettent aux étudiants d'obtenir un retour d'information immédiat qui leur donne l'opportunité de corriger leurs erreurs et rectifier directement leur compréhension du problème. Cela dit, ces modules de notation ont tendance à se concentrer sur l'exactitude du fonctionnement d'une solution aux dépends d'autres aspects du code. Cela réduit de façon significative la quantité d'information qui pourrait leur être transmise.

Dans ce mémoire, nous décrivons notre expérience de l'utilisation en ligne de la plateforme Learn-OCaml pour enseigner à des étudiants de deuxième année la programmation fonctionnelle dans le cadre d'un cours sur les langages et les paradigmes de programmation. De plus, nous explorons de quelle façon nous pouvons utiliser l'infrastructure de notation automatique de Learn-OCaml pour faciliter l'écriture de modules de notation plus expressifs. Ces modules, en plus de vérifier l'exactitude du code, permettent effectivement d'enseigner des éléments de style de la programmation fonctionnelle. En particulier, nous présentons nos extensions de la plateforme Learn-OCaml qui évaluent les étudiants sur la qualité de leurs tests et le style de leur code. De plus, nous décrivons comment nous avons utilisé ces extensions dans le cadre d'un cours pour faciliter l'enseignement de bonnes pratiques de programmation et faisons l'analyse de données provenant d'une étude sur l'utilisation de la plateforme par les étudiants grâce à la soumission de leur code.

À travers la présentation des leçons tirées de l'utilisation de la plateforme Learn-OCaml ainsi que des nouveaux outils et modules de notation que nous avons créés, nous désirons faire la promotion de l'éducation de la programmation fonctionnelle, améliorer l'expérience

éducationnelle des étudiants et rendre l'enseignement et l'apprentissage de la programmation fonctionnelle typée plus accessible à la fois aux enseignants et aux étudiants à travers le monde.

Acknowledgements

I would like to thank my supervisor, Prof. Brigitte Pientka, for taking me on as her student, and for her support, guidance, and patience over these past two and a half years. I couldn't have done this without your willingness to change my thesis topic a year and a half in, and jump into a new field that we were both unfamiliar with, taking this experiment with Learn-OCaml further than I ever thought it could go.

I would also like to thank my fellow COMP 302 teaching assistants whose work was crucial in making our experience using Learn-OCaml a success: Akshal Aniche, Ivan Miloslavov, Jacob Errington, Kelvin Tagoe, Nada Marawan, Scarlett Xu, Wilson Cheang, Yingzi Xu, and most of all Ariella Smofsky, who put an immense amount of work into her role as head TA and in helping me design our automated graders. Prof. Prakash Panangaden also played a large part, adopting the Learn-OCaml platform for use in his own offering of COMP 302.

My lab mates: David Thibodeau, Jacob Errington, Nathaniel Bos, and Wilson Cheang, and our visiting postdoc Szilvia Varró-Gyapay, were great sources of ideas and general advice on navigating grad school life. David also translated my abstract into French for me.

Finally, I would like to thank my friends: Andrea, Andrew, Gary, Javier, Keith, Lindsay, Mike, Renee, Ryan, and Simon, and my family, for their constant encouragement and support as I despaired that I would never be able to complete this thesis. I did it!

Contents

Chapter 1

Introduction

1.1 Automated grading

As enrollment in computer science programs continues to surge in universities across North America [Camp et al. 2017], more and more instructors are turning to software systems to grade their students' programming assignments. Not only does automating the grading process save a significant number of instructor and TA hours; it also removes the potential for grading mistakes and inconsistencies that occur when marking hundreds of assignments by hand, and allows for a much quicker turnaround time for students. When used effectively, automated grading has the potential to greatly enhance student learning experiences.

References to automated grading in the literature can be found as early as 1960 by Hollingsworth [1960], who decribes how student programs written in machine language were input on punch cards and run through a series of predetermined input/output tests. While the grading machine was not fully automatic, Hollingsworth states that it incurred significant time savings: "In general only an eighth as much computer time is required when the grader is used as is required when each student is expected to run his own program, probably less than a third as much staff time, and considerably less student time"; in addition, he asserts that students learned programming "probably better" than they did with the previous system

of manual grading, though this claim was at the time not backed by evidence.

The use of automated grading in higher education became much more popular near the turn of the century, with the development of systems such as TRY [Reek 1989], Ceilidh [Benford et al. 1995] (later CourseMarker [Higgins et al. 2003]), ASSYST [Jackson and Usher 1997], Scheme-robo [Saikkonen et al. 2001], BOSS [Luck and Joy 1999; Joy et al. 2005], and Marmoset [Spacco et al. 2006]. These systems generally made use of some form of electronic submission to a grading server which would compile the student's code, run it through some instructor-defined tests, and output the results in some form that could be relayed back to the student. Submission methods ranged from lower-level command-line scripts, to submitting by email, to using a plugin in an IDE such as Eclipse, to using a full-fledged GUI client with integrated course management facilities.

The literature around these automated grading systems was largely concerned with their *implementation*, with the following concerns being commonly cited:

(1) Privacy of the students' submitted files and resulting grades;

(2) Data integrity: making sure that student code was not lost or tampered with;

(3) Sandboxing the execution of untrusted code, to prevent malicious or broken student code from causing harm to the system or gaining unauthorized access to system data.

As these sorts of implementation issues were not as well explored at the time as they are today, much effort was expended in documenting these concerns and detailing how security features of the UNIX system, and later of the higher-level programming languages used for implementation, were used to address them. Meanwhile, relatively little emphasis was placed on the pedagogical implications of the assessment criteria used by these automated graders, and the quality and usefulness of the feedback they provided to students.

However, automated grading sytems could potentially have a huge positive impact on student learning. Some of the above automated assessment systems allowed for *repeated submissions*, meaning that after students received their results from the grading server, they

could modify their code and then resubmit it. This process could be repeated a number of times (possibly up to an upper limit specified by the instructor). Typically the grading script would provide students with feedback about the mistakes in their code preventing them from achieving full marks: for instance, the output from a failed test case, or an error message raised by their program. This allowed students to iteratively develop their programs and address errors in their understanding as they came up, according to the feedback provided by the grader. Having access to immediate feedback on their programs has been recognized to significantly improve student learning outcomes and engagement with their assignments (see, e.g., [Sherman et al. 2013; Wilcox 2015; Gramoli et al. 2016]). This potential remained sorely underexamined and underutilized in the early decades of automated assessment sytems.

1.2 Online programming platforms

Douce et al. [2005] outline the evolution of automated assessment systems over the past several decades through three broad "generations". The first-generation systems were the early attempts at automated grading such as [Hollingsworth 1960], which generally required a lot of specialized expertise to develop and use, and involved making modifications at the level of the compiler or the operating system itself. Second-generation systems, which the authors describe as "tool-based", were designed to be more accessible for instructors and students to use, and tended to involve the use of command-line scripts or GUI applications to submit assignments; the previously-cited assessment systems all initially fell into this category. The third generation represented a move to web-based technologies for delivering and submitting assignments, and these systems generally included course management features as well. Examples of this group of assessment systems include RoboProf [Daly 1999], the re-implementation of BOSS [Joy et al. 2005], and, as implied by the name, Web-CAT [Edwards and Perez-Quinones 2008].

A new trend in this age where distance learning and MOOCs (massive open online

courses) have been growing increasingly popular are systems that we will refer to as *online programming platforms*, in which an automated assessment system allowing repeated submissions is combined with an environment for developing code that runs entirely within the web browser. Examples of online programming platforms include the environment used by the popular learning platform Codecademy [Sims and Bubinski 2011], and the Haskell programming tutor Ask-Elle [Gerdes et al. 2017]. Typically these platforms offer a range of built-in programming exercises for users to attempt. These exercises are presented in an interactive environment where the user can read the description of their task, write their code in an integrated editor, evaluate their code, run the automated grader, and view the resulting feedback, all in one place. The user's progress and code are usually saved on the system's web server, so that they can resume from any other machine that has internet access and a web browser.

Online programming platforms offer immense potential to enhance students' educational experience in courses with high programming content. In addition to the previously-mentioned benefits of immediate feedback and allowing iterative code development, having an entire development environment just a click away in a browser lowers the entry barrier drastically and makes programming accessible for students of all computing backgrounds. 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 a particularly important consideration 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: OCaml, in particular, is very difficult for students using Windows to install, and the commonly-used editor for OCaml code is Emacs, which many early computer science students tend not to feel comfortable using.

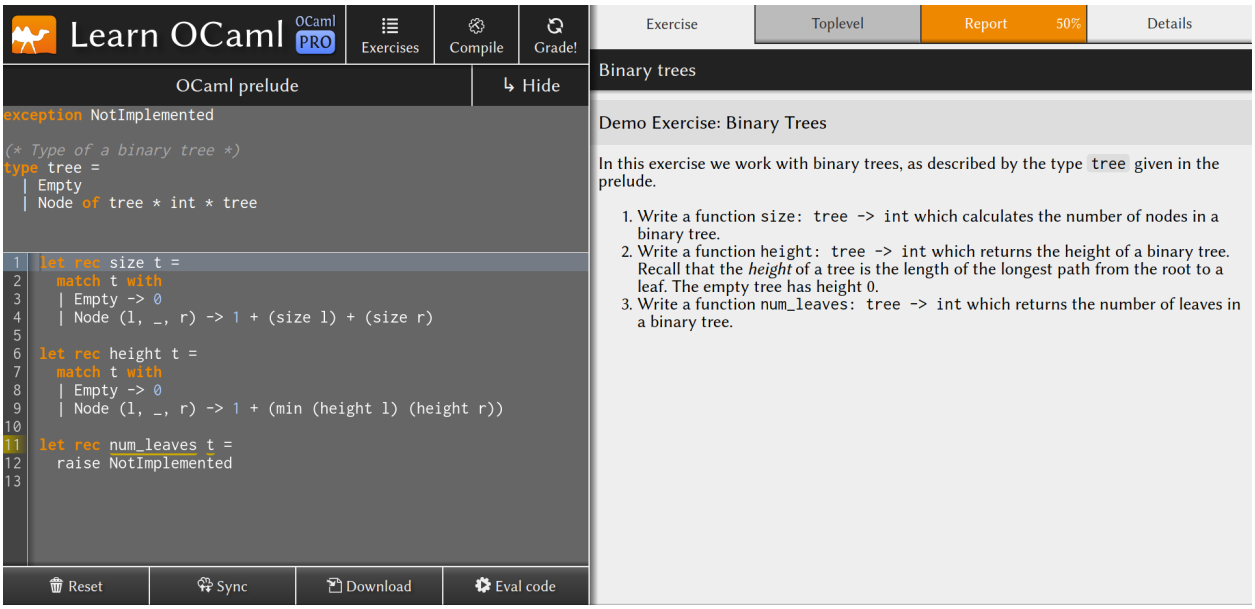
1.3 The Learn-OCaml platform

Since 2015, Paris Diderot University has administered a MOOC called *Introduction to Functional Programming in OCaml* [Di Cosmo et al. 2015], now colloquially known as "The OCaml MOOC". The course introduces learners with some programming experience to key concepts of statically-typed functional programming over the span of a six-week period. It has been a great success, with over 7000 students from more than 120 countries enrolled over the 2015 and 2016 sessions, of which 2418 were active in the course and 588 obtained a certificate of completion [Canou et al. 2017]; data from the 2017-2019 sessions has not been published. A significant reason for this success, according to student feedback, is the interactive exercise environment that was developed specifically for the MOOC. This exercise environment was later released as an open-source standalone online programming platform called Learn-OCaml [Canou et al. 2016, 2017].

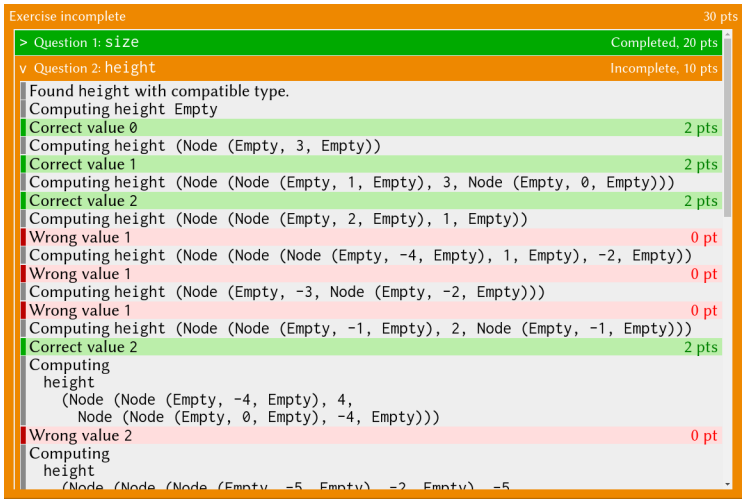
The Learn-OCaml platform offers students access to an interactive OCaml toplevel, two styles of tutorials (interactive and non-interactive) that can be written and uploaded by the course instructor, and a set of instructor-developed programming exercises. In this work we focus exclusively on the exercise functionality, although we briefly touch upon the tutorials in our discussion of future directions in Chapter 6.

A view of the exercise functionality as seen by students using the platform is shown in Figure 1.1a. In the right pane the student is given a description and instructions for the exercise. In this particular exercise, the student is asked to implement some basic functions on binary trees to measure their size, height, and number of leaves. On the left is a text editor, complete with line numbers, syntax highlighting, and automatic indentation, in which the student writes their solution. Compiler errors and warnings are underlined in red and yellow, respectively, with the line numbers highlighted in the gutter as in line 11 in the figure. Hovering over the highlighted portions reveals the error or warning message. Above the editor is a section containing optional *prelude* code, such as predefined datatypes and functions, which is loaded before the student code; for this exercise, the prelude is used to

define an exception `NotImplemented` and a datatype for binary trees. The left side of the screen also contains buttons to compile and typecheck the student's code; run the automatic grader on their solution and see the results; reset the contents of the editor to the original code template provided by the instructor; sync their code to the Learn-OCaml server; download a copy of their code; and run their code in an OCaml toplevel.



(a) Student exercise view



(b) A sample grader report

Figure 1.1: The Learn-OCaml interface

Automated graders for each exercise are written by the course instructor in OCaml, using

the high-level grading combinators and lower-level grading primitives provided by Learn-OCaml's testing library. These graders produce *reports* of the format shown in Figure 1.1b: organized as a hierarchy of foldable sections, with completed sections marked in green and automatically collapsed, while incomplete sections are expanded and errors are highlighted by colour-coded messages.

The report in Figure 1.1b was generated by running a simple automated grader on the student code shown in the editor pane of Figure 1.1a. The grader was written using a high-level function from the Learn-OCaml grading library to randomly generate inputs for each function to be tested, call the student's implementation of each function on the randomized inputs, and compare the output to that of a model solution. The student's implementation of the `size` function is correct, and thus receives full marks. The implementation of `height`, however, is incorrect, since the recursive case erroneously builds on the height of the smaller subtree instead of the larger one, and thus the function only returns the correct output on trees where every path to a leaf has the same length. This issue is exposed by randomized testing and the student receives only half marks for their implementation, along with some examples of inputs on which their function fails to return the expected output. It is then up to the student to debug the issue, correct their implementation, and resubmit their code for grading.

One might remark that this grader is concerned solely with the input/output correctness of the student's solution, and indeed this has been a general trend with automated grading. Many of the automated grading systems referenced in Section 1.1 focused heavily or even solely on input correctness, 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 or good coding practices to solve a particular problem, or even verify a less nebulous criterion like the use of a particular language construct such as pattern matching. As a consequence, students miss out on an

entire class of valuable feedback that is extremely important for their learning.

The bulk of the high-level portion of Learn-OCaml's grading library focuses on input/output testing, with much of the other functionality being as of yet underdocumented and underdeveloped. Hence graders written for Learn-OCaml tend to take a form similar to the example grader described above. 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 (abstract syntax tree). With this functionality, one could easily write a grader to examine the student's code to determine whether they used a required language construct, or to check for uses of a particular name. Going further, having access to the AST opens up a number of possibilities for performing static checks on the student's code using OCaml's compiler front-end libraries.

1.4 Related work

The most notable work in making functional programming accessible to beginning students is undoubtedly the DrScheme project [Findler et al. 1997; Felleisen et al. 1998; Findler et al. 2002]. DrScheme (now known as DrRacket) is a graphical environment for the PLT Scheme (now called Racket) dialect of Scheme. It was designed as a *pedagogic programming environment*, incorporating features such as precise run-time error locations and clearer error messages, an algebraic stepper, a transparent REPL (read-eval-print loop), algebraic output syntax, a static debugger, and syntax checking with lexical scope analysis. Prior to the introduction of DrScheme, students learning functional programming with Scheme typically worked with shell-based or Emacs-based programming environments. Using the Handin Server plugin[1], DrRacket can be set up so that students can click a "Handin" button in the IDE to upload their code to a server, where some instructor-defined tests are run and the results sent back 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

[1] `https://docs.racket-lang.org/handin-server/`

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 dynamically-typed functional languages in the Scheme family popular in the university setting, but there exists no such platform for statically-typed functional programming languages in the ML family. DrRacket is also not an online programming platform; students need to download and install it on their local machines, although it boasts excellent cross-platform support.

WeScheme [Yoo et al. 2011] is a web-based programming environment for the Scheme and Racket programming languages that places special emphasis on allowing students to write interactive, graphical programs, and especially on making it easy for them to share those programs with others. Unlike Learn-OCaml, it does not include facilities for delivering or automatically grading exercises, although exercise templates can be distributed using the sharing functionality.

Pyret[2] is a programming language designed for introductory computing education, with a web-based editor heavily inspired and influenced by WeScheme. In a similar vein, CodeWorld[3] is a web-based programming environment for an educational variant of Haskell, including an extensive set of tutorials in a similar style to the Learn-OCaml tutorials. As with WeScheme, both environments support saving and sharing programs in the cloud, but neither supports automated grading of exercises.

Another direction in the body of work on automated feedback is the area of *intelligent tutoring systems*, which provide students with context-sensitive hints on how to proceed when they get stuck on a programming problem, automatically generated based on the code the student has written. Le et al. [2013] provide a review of traditional approaches to intelligent tutoring system design. Ask-Elle [Gerdes et al. 2017] is an intelligent Haskell tutor that supports the stepwise development of Haskell programs by allowing students to

[2] `https://www.pyret.org/`
[3] `https://code.world/`

insert holes in their programs, which they can incrementally refine based on feedback from the tutor. Feedback is provided by the instructor in the form of annotated model solutions; when a student's code deviates too far from any of the known solutions, the tutor falls back on randomized input/output testing. A recent trend in other intelligent tutoring systems is using a data-driven approach in which hints are generated from other students' solutions [Barnes and Stamper 2008].

Other automated assessment systems have included facilities for automatically grading students on how well they test their own code. In these systems, code coverage is typically used as a metric for measuring the quality of a student's test suite. Edwards [2003a,b] observes that automated grading systems that focus heavily on code correctness lead students to neglect other aspects such as good design, proper use of abstractions, and appropriately testing their code. He proposes a grading system in which it is the responsibility of the student to demonstrate the correctness of their own code by providing a set of tests. Submissions are graded based on test validity (correctness of the test cases with respect to the assignment specification), test completeness (code coverage with respect to the reference solution), and code correctness (with respect to the student's own test cases only). This grading strategy is implemented in the Web-CAT automated assessment system [Edwards and Perez-Quinones 2008]; Edwards reports on the experience of using Web-CAT in [Edwards and Pérez-Quiñones 2007]. Chen [2004] reports a similar experience as Edwards with regard to the effect of using an automated grader that heavily emphasizes code correctness; he addresses this by requiring students to write test cases and grading them using a variant of mutation testing [DeMillo et al. 1978].

Another dimension of grading students' tests involves applying them to other students' implementations to supplement the instructor's test cases and to evaluate how many bugs the student tests are able to uncover. Wrenn et al. [2018] provide a review and evaluation of different models for using student tests to help judge the correctness of other students' implementations. In a similar vein, the CaptainTeach environment [Politz et al. 2014] requires

students to follow a multi-stage development process for each program they write, in which they first write a test suite for the program, then review some other students' test suites and provide feedback; only after this may they begin working on an implementation.

The question of what constitutes "good programming style", and how this could be quantified, has historically been a subject of active debate. Oman and Cook [1990] proposed a taxonomy for categorizing various facets of programming style; they suggested that this can be used as a basis for evaluating style, and as a framework for teaching students general principles of good programming style. In 1982, Rees [1982] presented STYLE, an automatic assessment tool for evaluating style of Pascal programs. Assessment was based on simple and easy-to-calculate syntactic metrics such as the amount of comments, length and variety of variable names, whitespace usage, and number of labels and `goto`s. Berry and Meekings [1985] built on this to create a style analyzer for C, which served as a basis for style checking functionalities in larger automated assessment systems such as Ceilidh [Benford et al. 1995] and ASSYST [Jackson and Usher 1997]. Ala-Mutka et al. [2004] developed a style analyzer for C++ based on the marking scheme proposed by Rees, which was used in several programming courses with favourable response from students. It was also integrated with Ceilidh [Benford et al. 1995] to combine style checking with evaluation of a program's functional correctness.

On the functional programming side, Michaelson [1996] develops a notion of *semantic* programming style, as opposed to *syntactic* programming style which is the focus of the previously-mentioned style assessors. While syntactic programming style is concerned with criteria such as identifier names and use of whitespace, Michaelson characterizes semantic programming style as "the use of appropriate [language] constructs": for example, using pattern matching as opposed to explicit data selectors, and using boolean values directly instead of comparing them to `true` or `false`. He develops a style checker for SML based on a list of AST rewrite rules, which suggests semantic-preserving transformation to students' code for better style.

Other tools exist that suggest code refactorings in the sense of Michaelson's style checker. Examples of these include Tidier, for Erlang Sagonas and Avgerinos [2009], and HLint[4], for Haskell. Tidier functions simlarly to Michaelson's 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 Michaelson proposes as well. The set of checks can be extended with custom rules added via a configuration file.

Following a method similar to the data-driven approach used by some intelligent tutoring systems, Choudhury et al. [2016] developed a system for functions written in Python that clustered student submissions based on AST similarity to group students that used the same strategy to solve a particular problem, and then allowed instructors to annotate these clusters with a style rating and guidance for the student. Once the system was seeded with this data, future student submissions could be automatically assigned to a cluster and given guidance accordingly. The system also automatically generated hints on language structures that could be added to or removed from a submission to improve its style, based on calculating a path from the student's submission to one of the solutions considered the "best". In a randomized controlled trial experiment, the authors found that students using the system showed a statistically-significant greater improvement in their code style compared to students given only a style guide written by the course staff.

1.5 Contributions

At McGill University, many computer science students take the course *COMP 302: Programming Languages and Paradigms* in their second year. 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 has historically been taught in a

[4] `https://github.com/ndmitchell/hlint`

variety of functional programming languages including Scheme, F#, and SML, and is now taught primarily in OCaml.

Enrollment in COMP 302 has increased nearly tenfold over the past several years: the course has grown from only 38 students in the Fall 2006 term to a record enrollment of 347 in Fall 2018. A history of enrollment numbers since 2006 can be found in Figure 1.2. It is particularly noteworthy that twice in recent years (Winter 2017 and Winter 2019) the number of students in the course at the end of the term has exceeded the set enrollment cap due to high demand.

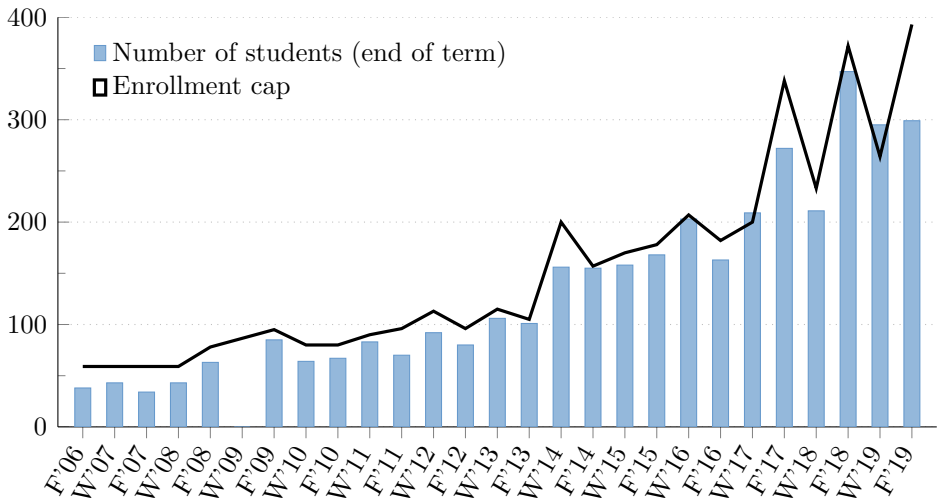


Figure 1.2: COMP 302 enrollment since 2006

As the number of students in the course has grown larger and larger, it has become less and less manageable to grade all of their assignments by hand. Grading takes up an increasingly large percentage of the teaching assistants' workloads, and the turnaround times become so long that the benefit students derive from the graders' comments is severely diminished: by the time they receive feedback on their code, students are often busy working on a later assignment and are no longer as interested in the previous ones. Occasionally students will continue to make the same error on many assignments in a row before being given feedback on this error on the first one.

To combat this problem, we began using the Learn-OCaml platform to deliver and grade

students' programming assignments in COMP 302. We piloted the use of the platform in our course in the Fall 2018 term. To our knowledge, we are the first institution to use Learn-OCaml in a university classroom environment. This brings with it a different set of challenges than one would encounter automatically grading students' assignments in a MOOC, where results are typically simply "pass" or "fail".

In this thesis, we report on our experience in using the Learn-OCaml platform in COMP 302 over the Fall 2018, Winter 2019, and Fall 2019 academic terms. 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; organizing and re-using code with modules and functors; 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. Our concrete contributions 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 (Chapter 2). We drew several lessons in writing good automated graders as we gained more experience with how students used the platform and implemented our own extensions to Learn-OCaml; we discuss here also how we applied these lessons in the Fall 2019 term.

(2) We release a repository[5] of homework problems and associated graders used in our course, available upon request, for use by other instructors wishing to incorporate Learn-OCaml into their curriculum. We envision this repository as a sort of online textbook, providing instructors with homework problems and their solutions, with the hope that others using Learn-OCaml might also contribute their own assignments and graders.

[5] `https://gitlab.cs.mcgill.ca/teaching-fp/learn-ocaml-repository` — access can be granted by contacting the authors.

(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 (Chapter 3), employing a strategy very similar to that of Chen [2004]. 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. It has been merged into the official distribution of Learn-OCaml.

(4) We present a set of extensions we have written for Learn-OCaml's grading library (Chapter 4). In particular, to teach students about good functional programming style, we release an extensible framework that supports style checking and provides suggestions (Chapter 4.3), heavily inspired by the SML style checker by Michaelson [1996].

(5) We evaluate the effectiveness of the above extensions to the Learn-OCaml platform by presenting the data collected when they were deployed in the Fall 2019 offering of the course (Chapter 5). We discuss student perceptions of the new tools, trends we discovered in how students were using the automated graders, and how these behaviours changed over the course of the semester.

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.

Chapter 2

Learn-OCaml and Our Curriculum

In this chapter, we describe how we have used the built-in features of the Learn-OCaml platform for delivering homework assignments in the Fall 2018, Winter 2019, and Fall 2019 terms. 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 Chapters 3 and 4. We later deployed these extensions and made use of them in many of our assignments in the Fall 2019 term; we discuss the changes we made to the course between iterations in light of our earlier experience in Chapters 3 and 4, and evaluate the impact of these extensions in Chapter 5.

2.1 Functional programming concepts

We have used the Learn-OCaml platform to grade a total of 23 homework assignments over 3 offerings of COMP 302. Here we highlight some of the main concepts these exercises tested and how we made use of the built-in aspects 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 In Fall 2018, 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 linked list of `card`s as in Figure 2.1, 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
```

Figure 2.1: 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.

For the last problem of this part of the exercise, students were asked to write a function to shuffle a hand of cards. As stated, this question is clearly not a good fit for automated grading, since one cannot tell simply by looking at the output of the function whether a list has indeed been shuffled in an unbiased manner. We addressed this by specifically requiring students to implement a modified version of the Fisher-Yates shuffle [Fisher and Yates 1948]:

(1) Let ℓ be the list of items to be shuffled, and let ℓ' be an empty list.

(2) Randomly select a valid index i from the list ℓ of items to be shuffled, using the built-in function `Random.int`.

(3) Remove the item x at index i from ℓ. Place x at the end of the list ℓ' of shuffled items.

(4) Repeat steps (2)-(3) until ℓ is empty. ℓ' is now a list of the elements of ℓ, in the order that they were randomly selected.

The high-level testing functions provided by the Learn-OCaml grading libraries allow us to specify some behaviour that should be carried out before a test case is executed. We used this to seed the random number generator with a constant before carrying out each test case, to ensure that the `Random.int` function would give the same outputs when called by both the student's implementation and the reference solution. It then sufficed to compare these outputs for structural equality. This was very simple to implement using the built-in grading libraries, taking only a few extra lines of code. A disadvantage of this approach was that it was difficult for students to debug their code if they had made a mistake in implementing the Fisher-Yates shuffle as described, since the output of their `shuffle` function would in many cases still appear to be a randomly-shuffled list. This is, however, more of a drawback of the question itself than of the grading format, since similar difficulties would present themselves when submitting an implementation to be graded manually: students would be unaware that something was wrong with their implementation until receiving their grade. All in all, we do not think that automated grading presented much of an issue with this exercise.

Higher-Order Functions Questions where we required students to use particular higher-order functions appeared, at first, to be relatively easy to test. A popular homework question in Fall 2018 (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 2.2. 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 those cupcakes that do not contain any of the items in `allergens`. They were required to implement this function using the higher-order functions `List.filter`, `List.for_all`, and `List.exists`.

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

Figure 2.2: Datatype definition of a cupcake made up of possible allergens

In most cases, we were able to verify that an implementation met those requirements fairly easily with a simple syntactic check on the code AST. The AST checking functionality of the grading library allowed us to easily find the binding for `allergy_free`, and traverse the AST of the body of the function to find calls to the required functions without needing to implement any of the traversal logic ourselves, or indeed even needing to understand very much about the representation of the parse tree.

However, this strategy of checking that a call to the desired function syntactically occurs within the function body raises both false positives and false negatives. Students soon discovered that they could get around this check by inserting spurious calls to the required functions within their code: for instance, they could assign the results of calling these functions to variables that they never used, or throw them away entirely; or they could call the required functions in a branch of a conditional expression that execution would never reach. On the other hand, some students wrote one or more helper functions outside the body of `allergy_free` in which they used some of the required `List` functions. The AST check did not detect these usages, since they occurred syntactically outside the body of `allergy_free`. This issue became more pronounced later on in the course, when students were more inclined to write their own helper functions.

One improvement we have made to the grader for future sessions is dynamically verifying whether the required functions are called during execution of the student's code. Learn-OCaml allows us to provide some code that should be executed before the student code or reference solution are loaded into the testing environment: this is similar to the prelude code,

except that it is hidden from the student. We used this to instrument the necessary functions in the `List` module to increment some internal references when called. By checking the values stored in these references during grading, we can verify if the student's code has indeed called `List.filter`, `List.for_all`, and `List.exists`. This allows us to give students credit for using the required functions in non-local helper functions, while giving no credit to students who include unused calls to these functions in sections of dead code. It does not, however, address the exploit of calling the required functions and then not using the results; and this approach does not work when we would like to enforce that students make use of a function they have defined previously in the exercise, since we do not have access to the function before testing in order to instrument it. We discuss this issue further in Chapter 4.

A second, more difficult problem, was when we did not want to specify which functions the students should use, but instead state that they should make appropriate use of higher-order functions in their code. For instance, we might want to tell students to look up the documentation for the OCaml `List` module and decide on the best higher-order functions to use for the situation. It is clearly infeasible to make a list of all possible higher-order functions a student could use to check against, and since the grading library gives us access to only the untyped AST, we have no other way of checking if a function used in the code is indeed a higher-order function.

We have attempted to get around this by instead forbidding the `rec` keyword that indicates a recursive function, so that implementations using manual recursion are not accepted. One drawback of this approach is that students can simply write a non-local recursive helper function and then call it within the function body: since the `rec` keyword does not syntactically occur within the body of the function, the grader is unable to detect it without writing a lot of extra code. Another, larger, issue is that we do not have a way of emphasizing which higher-order functions may be appropriate for a particular situation: for instance, that one should prefer more specialized functions like `List.map` or `List.for_all` over using `List.fold_left` to essentially re-implement those functions. This is an issue which is

difficult to address using automated grading; a possible solution would be to gather student submissions over several terms and identify common patterns which we could then program into the grader. We may, however, simply need to accept that this kind of problem is not well-suited to fully automatic grading.

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 2.3. 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 generally 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
}
```

Figure 2.3: 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.

References and State In Fall 2019, students were asked to implement two functions `add_head` and `remove` for doubly-linked lists, defined as in Figure 2.4. Each node contains some data and references to its previous and next nodes in the list. The main challenge for students in this exercise was that they were instructed not to duplicate any nodes in memory; so students had to be very careful when setting previous and next pointers, and needed to have a clear understanding of how values versus references to values are stored in memory to do well on the assignment.

```
type 'a llist =
  | Nil
  | Cons of 'a * 'a lcell * 'a lcell
and 'a lcell = ('a llist) ref
```

Figure 2.4: Definition of a doubly-linked list

Similarly to the previously-mentioned extra work required for grading "objects", the grader for this assignment involved a lot of code duplicated from the built-in grading libraries, since we were interested not in the outputs of the `add_head` and `remove` functions (which were simply the `unit` value), but in how these functions modified the list they were given. While the grading libraries include some resources for testing functions that mutate their inputs, these are specialized only for single-argument functions, and also do not allow us the flexibility to test sequences of operations on a single list.

However, working at such a low level gave us a lot of fine-grained control over the output of the grader, which we were able to use to our advantage. The default printed representation of cyclic data in OCaml is not very informative: cyclic references within a data structure are printed simply as `<cycle>`, without specifying what the cyclic reference is. An example of a doubly-linked list containing two elements, `"first"` and `"second"`, printed using this representation, is given in Figure 2.5a. It is impossible to tell simply from looking at this representation whether this list has been properly constructed without duplicating any data in memory, since this information is obscured by the `<cycle>` marker. We adjusted how doubly-linked lists, and in particular memory locations, were printed in the grading reports

```
{contents =
  Cons ("first",
    {contents = Nil},
    {contents =
      Cons ("second",
        {contents = <cycle>},
        {contents = Nil})})}
```

(a)

```
@1:{contents =
  Cons ("first",
    @2:{contents = Nil},
    @3:{contents =
      Cons ("second",
        @4:{contents =
          Cons ("first",
            <@2>,
            <@3>)},
        @5:{contents = Nil})})}
```

(b)

Figure 2.5: Printed representation of a doubly-linked list, by the REPL and by our grader

to make it easier for students to understand when nodes were being duplicated in memory; Figure 2.5b shows how the same doubly-linked list is printed by our grader. Reference cells are now tagged with numeric identifiers, and these identifiers are used to mark each cyclic reference. With this expanded representation, it is clear that data in this particular doubly-linked list has been duplicated in memory: the second node of the list contains a duplicate copy of the first node in its previous field (indicated by the memory reference labelled with the number 4). For comparison, the desired structure of this list is as follows:

```
@1:{contents =
 Cons ("first",
   @2:{contents = Nil},
   @3:{contents =
     Cons ("second",
       <@1>,
       @4:{contents = Nil})})}
```

Note that the previous field of the second node now points to the place in memory where the first node is stored, so that no data is duplicated.

On a short quiz following the due date for this exercise, the two multiple-choice questions relating to how data is laid out in memory were answered correctly by about 78% and 77% of students respectively. More data would be needed to make a solid conclusion, but this gives us a bit of a positive sign that, aided by the grader's feedback, most students were able to acquire a basic understanding of what is typically a very difficult concept in this course.

Control Flow In Fall 2018, students implemented a parser for a simple arithmetic language twice: once using exceptions for control flow, and once using continuations. These parsers were implemented as a set of mutually-recursive functions defined at the top level. Both implementations 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.

However, we had a different experience in Fall 2019, when we asked students to implement a function `find_path` for finding a path between two nodes in a graph, using first exceptions and then continuations for control flow. With the parsing example from Fall 2018, there was a very clear set of recursive helper functions to decompose the problem into: one recursive helper function should be implemented for each non-terminal symbol in the language's grammar. However, when it comes to the `find_path` problem the decomposition is not so clear, as there is no single "canonical" solution. This was an issue because we did not want to constrain students into following any particular problem-solving strategy by telling them how they should decompose the problem. We therefore could not pre-define any recursive helper functions at the top level for students to implement; students were told to decide on their own strategy and define recursive helper functions appropriately. This meant that for the implementation using continuations, the grader could not call the students' helper functions and thus the argument continuations were not exposed to the grader, so we could not test students' continuation use as we did with the parser in Fall 2018.

We attempted instead to test continuation use indirectly by strongly emphasizing that students should make use of continuations to make their implementation tail-recursive, and

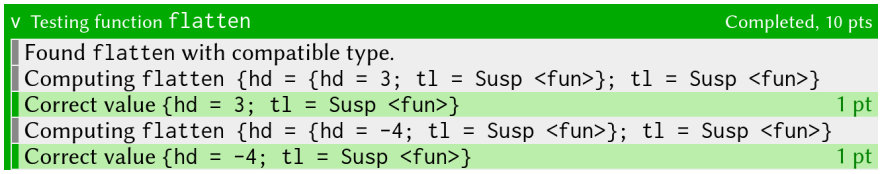
calling their function on a graph containing only a single very large path between two endpoints to check if this trigged a stack overflow. A proper tail-recursive implementation should use constant stack space and thus execute without issue even on this large input. Unfortunately, this did not work out very well in practice: Learn-OCaml uses the `js_of_ocaml` library [Vouillon and Balat 2014] to compile students' OCaml code to JavaScript that is then run in the browser, and JavaScript does not support tail call optimization. The `js_of_ocaml` compiler performs tail call optimization of some common tail call patterns, but even slight variations on those result in stack overflows, and some tail calls are not optimized at all. It turned out that this was far too unstable for our purposes: students would experience stack overflow errors on correct tail-recursive implementations, triggered by factors such as continuations that were not eta-reduced, or even just by using certain browsers. All in all, this did not provide for a useful learning experience, and in the future it would be best to stick with problems such as the parser with an obvious implementation strategy by which we can directly test students' use of continuations provided as argument.

Lazy Programming One of the final assignments of the Fall 2018 term focused on lazy lists, as defined in Figure 2.6. Delayed computation is modelled by a function that takes an input of type `unit`; lazy 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.

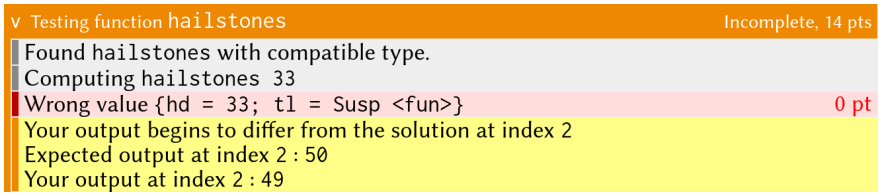
```
type 'a susp = Susp of (unit -> 'a)
let force (Susp s) = s ()
type 'a lazy_list = {
  hd: 'a;
  tl: ('a lazy_list option) susp
}
```

Figure 2.6: Definitions for lazy computation

Students were also tasked with implementing a function `hailstones` that lazily generates



(a) Default printed representation of a lazy list



(b) Output of our custom lazy list comparator

Figure 2.7: Grading reports for functions involving lazy lists

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 2.7a, 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. At the

moment, doing this requires writing a fair amount of extra boilerplate code, as Learn-OCaml does not provide a hook into its high-level testing functions for adding custom messages to the test reports based on the test inputs.

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 2.7b.

An unexpected situation arose due to some students' temptation to use references and fall back upon the imperative programming practices they were used to in order to solve some of the problems. Students attempted to think about their implementations in terms of list indices and imperatively looping through the elements of a list instead of by decomposing the structure of a lazy list. They did this by mutating a reference storing a current index within the delayed computations of their lazy lists. Thus upon traversing parts of their lists once everything appeared to be in order to the grader, but their functions were in fact incorrect: due to the side effects encoded in their delayed computations, accessing the tails of their lists a second time would yield incorrect results. We found that about 8% of final student submissions made use of the assignment operator (`:=`). For future iterations of this assignment we have forbidden the use of `:=` by redefining it to raise an exception stating that it should not be used in this exercise; we hope that this will be sufficient to deter students from trying to use mutation when we do not intend them to.

Modules In Fall 2019, we had students practice working with modules and functors by having them use modules to separate distinct units of measurement, so that the type system would raise an error if one unwittingly tried to combine, for example, operations on meters with operations on miles. Students wrote an implementation of the module type `METRIC`, as

given in Figure 2.8a, using floating-point numbers, and used this implementation to create representations for meters, kilometers, feet, and miles. They then implemented a functor `Speed`, parameterized by a `METRIC`, for performing speed calculations over a given metric; the module type `SPEED` of the instantiations of `Speed` is given in Figure 2.8b.

```
module type METRIC =
sig
  type t

  val unit: t
  val plus: t -> t -> t
  val prod: float -> t -> t
  val to_float: t -> float
  val from_float: float -> t
end
```

(a)

```
module type SPEED =
sig
  type s
  type distance

  val speed:
    distance -> Hour.t -> s
  val average: s list -> s
  val to_float: s -> float
end
```

(b)

Figure 2.8: Definitions of modules for working with metrics

It took us a lot of work to modify the original version of this assignment to one suited for automated grading. In order to ensure type safety, the high-level functions of the grading libraries require us to provide the expected type for each of the student's functions that we wish to call for testing purposes. These provided types must exist independently of the student implementation, meaning that we cannot provide the name of a type that the student is supposed to implement. We therefore could not directly test any module functions involving an abstract type using the library's high-level graders; even when the module provided conversion functions (such as `to_float` in the signatures `METRIC` and `SPEED`), we could not utilize these functions as the abstract type of course appeared in their type signatures.

Using the library function `test_module_property`, which gives us access to the student's module implementation as a first-class module if we can provide that module's signature, it would be possible to manually call and grade the student's implementation by using their conversion functions when necessary. There also exists a library function `test_student_code`, which can be used to package the entirety of the student's code file into a module which can then be tested similarly. However, using these functions would require us to re-implement a

lot of the functionality of the library's higher-level grading functions; and the error messages these functions provide, which are on the level of a module signature mismatch, tend to be quite verbose and confusing for students to read and understand. We chose instead to expose the type `t` in the implementation of `METRIC` that students were to fill in, so that we could test their functions with inputs and outputs of type `float`, and then required students to keep the type abstract for the metrics defined using this implementation by repackaging them with the `METRIC` signature. A library function allowed us to test each of their metrics to check that the type `t` was indeed abstract.

We ran into another issue when trying to test implementations of the `Speed` functor. Since functors are not first-class values in OCaml, we could not retrieve the `Speed` functor in our grader code as we could with a module. Exposing the types `s` and `distance` would also not have been enough, as the type `distance` was being implemented as the abstract type `t` of the students' miles and kilometers metrics. We instead added two functions `speed_as_float: float -> float` and `average_as_float: float list -> float` to the module type `SPEED` and pre-implemented these functions in the code template to compose with the relevant conversion functions, and added a line to the template instantiating the `Speed` functor with a dummy `METRIC` implementation. We were then able to grade this instantiated module as normal.

Finally, we wanted students to implement a module providing conversion functions between specific pairs of units, such as converting miles to kilometers. However, we ended up removing this question from the assignment since we could not find a way to grade it without inserting yet more boilerplate code into the template. Since the type of the module would involve types defined by the student in the assignment, there was no way for us to use any of the built-in high-level grading functions to access it. We would instead need to define several functions at the top level to perform the relevant type conversions before and after calling functions of the conversion module, trusting students not to modify them, which we ultimately decided against.

All in all, we ended up simplifying or removing several elements of this assignment in order to make it suitable for automated grading without needing to write a large amount of grader code. There was a lot of code that we would prefer for students to have to write themselves, such as module declarations, prewritten in the template in order to make sure the correct types were exposed to the grader, and that students would be able to compile and grade their partially-completed assignments without needing to comment out any of the given code. This somewhat diluted the purpose of the assignment, as the actual functions students needed to implement within their modules were themselves quite simple. In the future we may prefer to use the `test_student_code` function previously described, though it requires us to implement a lot of the grading infrastructure ourselves.

Programming Language Implementation At the end of each term, we give students one or more exercises in which they implement parts of a compiler or interpreter, such as type-checking, type unification, code generation, or evaluation, for a small toy language. Features of the language generally include integer and boolean values, conditional expressions, function abstraction and application, let-expressions, and recursion; additional concepts such as tuple construction and deconstruction, n-ary lambdas, and more sophisticated let-expressions are swapped in each term.

Writing graders for these assignments was very similar to the experience one might have writing an automated grader for a programming course without using a platform such as Learn-OCaml, as we did not make use of any additional features such as AST inspection or fully randomized testing. Our graders were simply written as a series of preset test cases on which we executed the student's functions and compared the output to that of the model solution (if necessary, after normalizing both into a form such as de Bruijn [De Bruijn 1972]). In Fall 2018 these test cases were completely hard-coded, and hidden from students in the grader output to prevent them from hard-coding these cases into their functions to obtain full marks. Students' reactions to the hidden test cases were generally negative, and many

expressed frustration with them in the end-of-term survey. Indeed, completely hiding the tests from students gives up many of the benefits of automated feedback that we are using Learn-OCaml for in the first place; so in the Fall 2019 term, we instead wrote a generator for each individual test case which would generate randomized expressions of the toy language with a certain desired structure: for instance a function abstraction where all bound variables are referred to in the function body, or a series of nested let-expressions in which a variable is shadowed. Writing these generators took, as expected, quite a bit of extra effort and lines of code, though we expect that most of them can be reused for future iterations of the assignment.

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, which we summarize here. Many of these issues have been previously encountered by instructors introducing automated grading into their courses, and indeed several of them seem blatantly obvious after the fact; but as they are not well-documented in the functional programming community, we expect that many instructors making the switch to automated grading in their functional programming courses would experience them as we did.

Writing Graders is Difficult and Time-Consuming In Fall 2018, 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.

Other assignments that may be unsuitable for automated grading are those that allow students a significant measure of creative freedom in their design. It is always tempting to allow students more leeway rather than restricting them to a particular implementation, but it is our experience that this does not go well with automated grading: we find that the programming problems that work best for automated grading are those with strict and rigorous specifications. Attempting to give students more freedom tended to result in graders that involved considerably larger amounts of code and were yet more brittle. For instance, asking students to write an implementation using high-order functions of their choice from a particular set, instead of restricting them to the exact ones that are used in a particular model solution, meant that we had to program a set of alternative solutions into the grader, and reject any others that students came up with without a clear reason.

It is also infeasible to keep coming up with graders for new assignment problems year after year. Therefore, we have started a repository of pre-written graders, starting with the ones we have written for the Fall 2018 and Fall 2019 terms, that can be re-used over the years. As a contribution of this thesis, we release this repository to other instructors teaching courses in OCaml upon request[1]. 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.

Circumventing Grader Requirements We found it quite difficult to enforce properties of students' implementations such as requiring or forbidding them to use certain functions, or banning the use of manual recursion, that could not be straightforwardly verified by input/output testing. However much effort we put into detecting violations of such requirements, enterprising students were able to find their way around them, whether it was by defining other helper functions at the top level or using a required function in a throwaway fashion to trick the grader. At some point, expending the additional effort to try to make the graders as foolproof as possible was just not worth the increasingly diminishing returns in trying to imitate what would appear an obvious exploit to a human grader.

In the Fall 2019 term, we decided to switch instead to a method of semi-automated grading (see, e.g., [Jackson 2000; Tremblay and Labonté 2003]), where we had TAs screen assignment submissions after the deadline to ensure that students were not skirting the requirements (intentionally or unknowingly), and post a summary of common mistakes students made after screening was done. With this change, we also shifted our focus from trying to account for all possible ways students could get around the grader requirements, to instead providing clear requirements and feedback so that students were unlikely to fail to meet requirements without being aware that they were doing so, and so that TAs could

[1] `https://gitlab.cs.mcgill.ca/teaching-fp/learn-ocaml-repository` — access can be granted by contacting the authors.

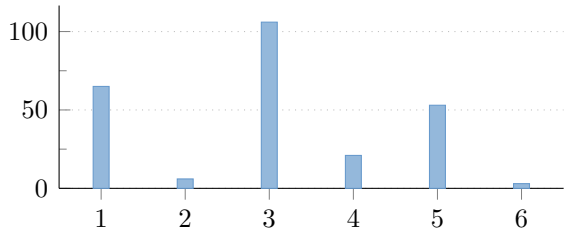


Figure 2.9: Number of submissions that received a grade deduction during TA screening, per assignment.

glean as much information as possible by quickly glancing at a student's grading report.

The number of submissions receiving grade deductions during TA screening varied wildly between assignments, as can be seen in Figure 2.9. We suspect that much of this variance is due to the fact that each assignment was screened by a different TA, and strictness of screening was not consistent between different assignments; thus we do not think any useful conclusions can be made from this data.

Student reception to the summaries of common mistakes that were posted after each screening were somewhat positive, according to an end-of-term survey. Several students were not aware that we had been posting these summaries at all, while others found the summaries too long and did not read them. A few students said that they had not made any of the mistakes described in these summaries, and so did not find them useful.

Improper Use of the Grader A major issue we experienced over the course of the Fall 2018 term was that students were becoming over-reliant 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 throughout 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 Edwards [2003b]; Chen [2004].

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. This is not possible to do in Learn-OCaml at the moment, as all grading takes place client-side in the student's browser, which makes such restrictions easy for an enterprising student to bypass. Creation of new accounts on the Learn-OCaml system is also unrestricted, so students can create multiple accounts to get around grading limits as well.

We decided to take a different route starting in the Fall 2019 term, 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] at the University of Michigan, 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 Chapter 3.

In addition, to make sure that students were aware of the existence of the toplevel and how to use it, we placed extra emphasis on the toplevel in the Fall 2019 term by using a newly-added "playground" mode of the Learn-OCaml platform in our lectures. This mode allowed instructors to write lecture code in an editor pane on one half of their screen, and send their code to an interactive toplevel pane on the other half of the screen to be evaluated, similar to the setup an OCaml programmer might use while editing OCaml code in Emacs.

We evaluate the impact the above measures had on our students in Chapter 5.

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 Chapter 4 and evaluate in Chapter 5.

Chapter 3

Encouraging Students to Write Tests

As mentioned in Chapter 2.2, a drawback of having a readily-accessible automatic grader for our homework assignments was that it was tempting for students to rely solely on the grader to tell them if their code was correct, instead of testing their own code thoroughly. This led to students treating assignments as a game of trial and error between themselves and the grader, in which they would repeatedly make small changes to their code and rerun the grader to "see what happens". Anecdotally, it was very rare for us to see students testing their own code in the OCaml toplevel.

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. The approach used by our extension is to assess the quality of a student's tests using a strategy based on *mutation testing* DeMillo et al. [1978].

3.1 Mutation testing

Mutation testing is a method of testing used in the software development industry to evaluate the quality of a test suite and design new tests with which to extend it. In short, mutation testing works by introducing bugs into the programmer's code, and then running the test

suite on the modified code to verify whether the tests are able to detect the newly-introduced bugs. If the modified code, known as a *mutant*, continues to pass all of the tests, then the test suite is deemed insufficient, and the programmer extends the test suite with one or more new test cases designed to expose the issue. If the mutant is flagged by the tests, the test suite is said to have *killed* the mutant. The quality of the test suite is then assessed by the percentage of mutants that are killed during the process of mutation testing.

For example, consider the implementation of a simple exponentiation function `pow` that works on non-negative integers, such that `pow n k` evaluates to n^k for all non-negative integers n and k (we do not care what `pow 0 0` returns). Suppose that we have access to an exponentiation operator `**` that does exactly this, so a programmer implements the function simply by writing:

```
let pow n k = n ** k
```

The programmer then designs a set of test cases for the `pow` function as in the following table: she asserts that `pow 4 1` should evaluate to `4`, `pow 0 5` should evaluate to `0`, and `pow 2 2` should evaluate to `4`. As these test cases are indeed true to the specification of `pow`, and the implementation of the `pow` function correctly adheres to the specification, her function passes all of the test cases.

n	**k**	**Expected**	**Actual**
4	1	4	4
0	5	0	0
2	2	4	4

Now, however, suppose that the programmer has made a small mistake when writing the definition of `pow`, and has instead written:

```
let pow n k = n * k
```

Notice that instead of the exponentiation operator (`**`), the programmer has accidentally written the multiplication operator (`*`).

This implementation of `pow` does not meet the specification, but it still passes all of the programmer's tests: the test suite is not strong enough to be able to distinguish multiplication from exponentiation. In order to improve the quality of her tests, the programmer adds a new test asserting that `pow 2 3` should evaluate to `8`. Since `2 * 3` evaluates to `6` and not `8`, her test suite now reports an issue with her implementation.

`n`	`k`	**Expected**	**Actual**
4	1	4	4
0	5	0	0
2	2	4	4
2	3	8	6

Mutant generation is typically done automatically by mutation testing frameworks such as the PIT [Coles et al. 2016] framework for Java, or MuCheck [Le et al. 2014] for Haskell. These frameworks make small changes to the programmer's code (or the generated bytecode, in the case of PIT) by applying *mutation operations* such as replacing a variable reference with a constant, negating a boolean expression, or replacing one mathematical operator with another. The idea behind this, as originally presented by DeMillo et al. [1978], is the *competent programmer hypothesis*, which posits that (professional) programmers create programs that are close to being correct, and most faults they introduce into their programs are caused by small errors in syntax. The intent of applying mutation operators, then, is to imitate common programming errors.

Unfortunately, for mutation testing to be effective, a large number of mutants need to be generated, and many of these mutants may be superfluous. The process of generating such a large amount of mutants and running tests over them is computationally very expensive for a codebase of non-trivial size. Increasingly more attention has been devoted to techniques for reducing the cost of mutation testing, for instance by generating mutants more selectively; for a survey on developments in the field, one can consult [Jia and Harman 2011].

3.2 Automatically grading students' tests

Our extension to the Learn-OCaml platform uses a variation on mutation testing to grade students on the quality of their tests. Suppose that we wish the student to write tests for the function `pow` as specified above. The student provides a set of test cases as a list of input/expected output pairs, as shown in Figure 3.1a. The set of tests indicated in the Figure is the same as initial test suite given in the previous mutation testing example.

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

(a) A set of sample unit tests

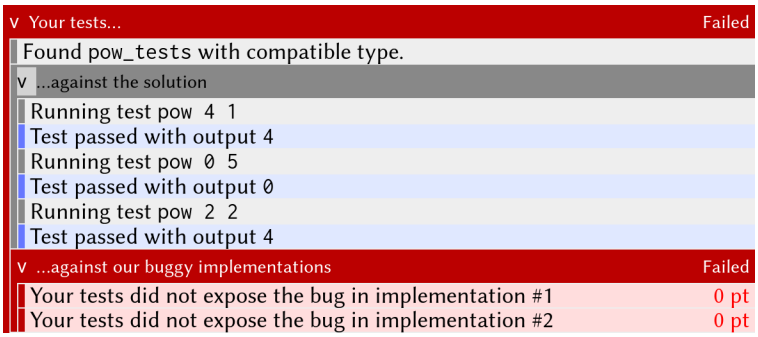
```
let pow n k = n * k

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

(b) A set of sample mutants

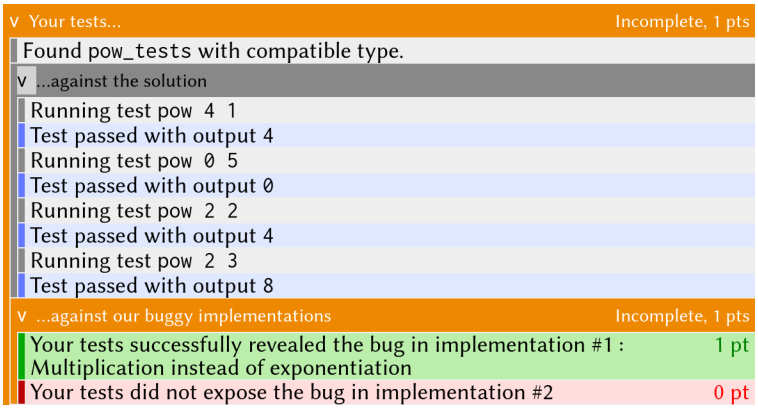
Figure 3.1: Using our mutation testing extension in Learn-OCaml

Programmed into the grader is a set of mutants of some reference implementations of the `pow` function, as given in Figure 3.1b. The first mutant is the one previously discussed, which multiplies its inputs instead of performing exponentiation. The second has a typo in the consequent of the first `if`-expression, returning `0` instead of `1` in the base case of exponentiation by zero. When adding a mutant to the grader, the instructor chooses a text description for the mutant that will be displayed once it has been killed, as well as how many points students should receive for killing that mutant.

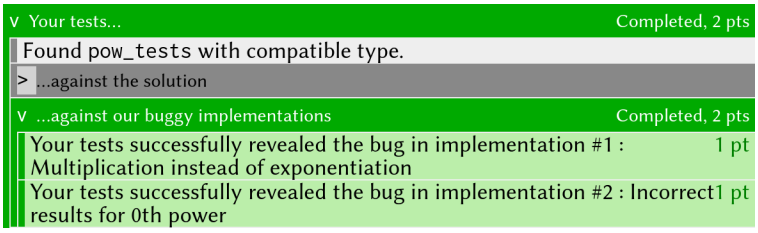
When the grader is run, it first checks the student's tests for correctness by running them against the correct solution code, as seen in the first section of the report in Figure 3.2a. Once all of the student's tests have been deemed correct, the test suite is next evaluated for quality by running it against each of the given mutants. The feedback report, as seen in the second section of the report in Figure 3.2a, informs the student of how many mutants her test suite has killed, and how many her test suite was not able to detect. The student may then revisit her tests and think harder about what cases she may not be covering, and extend her test suite accordingly. Figure 3.2b shows the resulting report when the student



(a)



(b)



(c)

Figure 3.2: Grading reports generated by our mutation testing extension

adds the test case `((2, 3), 8)`, asserting that `pow 2 3` should evaluate to `8`: the test suite is now able to kill the first mutant, but not the second. Finally, the student may realize that she has not included a test for the base case, and add the test case `((4, 0), 1)`; the resulting report (with the first section of the report collapsed for space reasons) is shown in Figure 3.2c.

Optionally, instructors may choose for the grader to also run the submitted test suite on the student's own code, with these results displayed in a separate section of the report.

Compared to the software development setting, in which a large number of mutants are

automatically generated based off of the programmer's implementation to see what kinds of bugs their test suite can uncover, we already know what a desired solution looks like, and what kinds of inputs we want our students to test. We can thus write a small number of very specific mutants manually while writing our graders, 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.

3.3 Our mutation testing extension in practice

We deployed our fork of the Learn-OCaml platform along with the mutation testing extension for use in the Fall 2019 offering of COMP 302. Having students write their unit tests as input/output pairs was simple enough that we could have them start writing tests straight away on their very first homework assignment, given a proper template.

Aside from the obvious purpose of using the framework to have students write tests for their own functions, we also used the mutation testing functionality to have students write test cases for pre-implemented functions that they would later be using in their code. For instance, in one exercise we provided students with a higher-order function `tabulate` such that `tabulate f n` evaluates to the list `[f 0; f 1; ...; f n]` (in later versions of OCaml than the one used by the Learn-OCaml platform, this is included in the standard library as `List.init`). Before using it in their code, we asked students to write a few test cases for `tabulate` to confirm that they understood how the function worked. We supplied the grader with two mutants: one that gave incorrect results for `n = 0`, and one that produced the list `[f 0; f 1; ...; f (n - 1)]` for non-zero inputs `n`. This ensured that students were able to correctly provide the output of `tabulate` in an example of each of these cases before they proceeded to use it in later questions. In previous years students have had issues with off-by-one errors when using the `tabulate` function, since it, possibly unintuitively, produces

a list of length `n + 1` rather than of length `n`. As this time students needed to explicitly confirm their understanding of the output before using `tabulate`, we did not encounter such an issue with this exercise.

We attempted to use the mutation testing functionality to require students to write tests for their own functions as often as we could. Unfortunately, there were some cases in which writing unit tests using the framework did not make sense, or where students' test-writing capabilities were restricted. Since students' test cases are provided in (homogenous) lists, they were restricted to a single concrete type for polymorphic functions: for instance, tests for a function of type `'a list -> 'a` would be restricted to a single instantiation of the type, such as testing only instances of the function at type `int list -> int`. This was somewhat concerning as we have historically had students submit functions that they had accidentally made monomorphic, not understanding that there was an issue because the example inputs we provided were all only for a single instantiation of the function's type. While we were able to verify that students' functions were indeed polymorphic when testing their actual implementations, it would have been ideal to have students demonstrate that they understood how their functions should operate on differently-typed inputs.

Another issue caused by the fact that tests were encoded as homegenous lists of input/output pairs was that we could not ask students to write test cases for inputs that were expected to raise exceptions. In earlier assignments, this was not such a big deal because most of the functions students had to implement raised exceptions only when given inputs we were not interested in: for instance, a function computing factors of a positive number, which raised a `Domain` exception when given non-positive inputs. In these cases we were not interested in having students test these inputs at all. However, in later assignments the cases in which student implementations were expected to raise exceptions were just as significant as the non-error scenarios: for example, when implementing evaluation and type inference for a toy programming language, students were asked to raise specific exceptions in the case of runtime or type errors. In this assignment we would have liked students to be able to

submit test cases for these kinds of inputs themselves, instead of only having these cases tested by our own grader. It would be possible to modify the mutation testing framework to allow specifying tests for exception cases, by using the `result` variant type as defined in the Learn-OCaml testing libraries and writing functions `assert_ok` and `assert_err` to create test cases (in order to hide the extra complexity of this datatype from the student), according to the specification in Figure 3.3. This change would also slightly simplify the formula of test cases, as students would no longer need to write their tests as nested pairs when testing functions that take multiple inputs. We are strongly considering making this change for future terms.

```
type 'a result =          val assert_ok:
  | Ok of 'a                'a -> 'b -> 'a * 'b result
  | Error of exn          val assert_err:
                            'a -> exn -> 'a * 'b result
```

Figure 3.3: Modifying the mutation testing library to support testing error cases

Finally, there were some situations in which asking students to write test cases did not seem particularly instructive due to the nature of the inputs and outputs of the functions they were writing. An example of this is an assignment in which students were working with matrices (represented as nested lists) of floating-point numbers and using them to perform statistical calculations. Here we were not particularly interested in the exact numbers that students were calculating, but in how they were using higher-order functions to process lists of lists; and we did not think it very useful for students to manually calculate the desired output values in order to construct test cases. We thus did not require students to write test cases; we instead simply reminded them that they were expected, as always, to test their own code, and hoped that they did so.

As another example, in a later assignment we asked students to implement a function `find_path` which would return a single path with no cycles between two given vertices in a graph. Since there could be multiple valid outputs to a given call to `find_path`, it did not make much sense to ask students to provide test cases with specific outputs.

We were able to evaluate students' test suites for significant parts of 4 of the 6 homework assignments we gave in the Fall 2019 term. In each of these assignments we asked students to write some test cases for their functions *before* they began implementing them, although unfortunately the Learn-OCaml platform allowed us no way of enforcing that they should do so. We tried to further encourage students to write tests before implementing by hiding all information about our own tests performed on each student's code, except for the final grade, until the student achieved full marks on their test cases by uncovering all of our mutants with their test suite. This meant that students did not have access to our own tests until they had written sufficiently strong test suites of their own. Despite these measures, we found that a few students were implementing their functions first and then using those functions themselves to generate expected outputs for their test cases, which is something we are unable to prevent without being able to enforce the order in which students complete parts of their assignments.

We also encouraged students to make use of the testing infrastructure to experiment with how they thought the functions they were asked to implement should work, by writing test cases and then using the grader to verify their understanding against the solution. If the expected output of a student's test case did not match the actual output produced by the solution, the grader would alert them that their test was incorrect, although it would not provide them the actual expected output as given by the reference implementation. It would then be up to the student to go back and fix their test by rereading the assignment specification and thinking through what the actual output should be for their test case.

For each assignment, we carefully thought through the mutants that we included for each function. There are two main approaches we explored for creating mutants:

(1) Generating mutants by making small changes to one of the reference solutions, true to the original idea of mutation testing: we generally found that this method did not give us very useful mutants when we tried to follow it for our earlier assignments. Making small changes to the reference solutions tended to result in very unspecific mutants

```
let rec sum_up l =
  match l with
  | [] -> 0
  | x :: xs -> x + sum_up xs
```

(a) Correct implementation of `sum_up`

```
let rec sum_up l =
  match l with
  | [] -> 2
  | x :: xs -> x + sum_up xs
```

```
let rec sum_up l =
  match l with
  | [] -> 0
  | x :: xs -> x * sum_up xs
```

(b) Mutants created using method (1)

```
let sum_up l =
  match l with
  | [] -> raise Fail
  | _ -> Solution.sum_up l
```

```
let sum_up l =
  match l with
  | _ :: _ -> raise Fail
  | _ -> Solution.sum_up l
```

(c) Mutants created using method (2)

Figure 3.4: Mutants of a function for summing up the elements of a list

that produced incorrect results for every input or almost every input, as the functions we were asking students for these assignments were generally short, simple, recursive implementations.

As an example, consider using this method to generate mutants of a reference implementation of a function `sum_up` (Figure 3.4a) which calculates the sum of the elements in a list using manual recursion. Some such mutants are shown in Figure 3.4b. The first mutant incorrectly returns `2` in the base case instead of `0`, meaning that the function returns an incorrect result for all possible inputs; this is clearly not a useful mutant, as it can be exposed by any test case whatsoever. The second mutant incorrectly performs multiplication instead of addition; combined with the (now correct) base case, this means that the mutant will return `0` for all inputs. This mutant is somewhat more useful, as it can be exposed by any non-empty list in which the sum of the elements is *not* zero, but this extra side condition is not particularly intuitive to students, and is not a particular class of input that we are very interested in having students test specifically.

(2) Generating mutants by deciding on a particular class of input, and then writing func-

tions that raise an error upon encountering that exact class of input, while falling through to the reference implementation to process all other inputs: this was the method that we preferred to use throughout the term. While these function definitions were contrived and not actually "mutants" in the original sense of the word, they allowed us to express exactly what classes of inputs we wanted students to test, and write mutants that would fail on *only* those inputs. For example, some mutants generated from the implementation in Figure 3.4a are given in Figure 3.4c: the first mutant fails on the empty list, and the second mutant fails on all non-empty lists, precisely specifying the two categories of inputs we would like students to test. If, in addition, we wanted to make sure students were testing a list with at least two elements, we could easily write a new function that fails only on lists with more than one element in the same fashion.

As calling the mutants generated by method (2) "buggy implementations" is somewhat misleading to the students, and these functions are no longer really "mutants" at all, we will likely rethink how we present this testing functionality to students in the future.

We discuss how students used the mutation testing framework, and evaluate how useful it was to their learning, in Chapter 5.

Chapter 4

Writing More Powerful Graders More Easily

As it stands, the grading library provided by the Learn-OCaml platform has a lot of potential for writing powerful graders, since it allows graders to access the student's code AST. However, the library only provides helpers for carrying out simple and context-free 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 to perform more sophisticated checks on the syntax tree, and this code needs to be copied and pasted into each grader separately. There have also been 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. We have therefore written a new module for the grading library that provides graders with an augmented version of the parse tree, annotated with typing and scoping information. Over the course of our first term using the platform, we used this module to write several extensions 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. We describe our annotated parse tree structure, and our

extensions implemented using it, in this chapter.

A general theme that has been mentioned before, and will come up multiple times in this chapter, is that our automatic analyses are neither sound (raising no false negatives) nor complete (raising no false positives), nor do we find it worthwhile to strive to achieve either of these properties. As discussed in Chapter 2.2, we have found that some students will work very hard and come up with very inventive solutions to bypass grader requirements on the structure of their code. We consider seeking to cut off all possible exploits to be misplaced effort, and instead advocate for a *semi-automated grading* strategy where automated grading reports are supplemented by manual human screening. False negatives are more concerning, but we have found it easy enough to design our assignments so that specific grader extensions are not used in contexts where false negatives may come up.

4.1 Implementation of Grader Extensions

The Learn-OCaml grading libraries give us access to the student's untyped AST in our grader code, represented using the datatypes from the OCaml compiler's internal `Parsetree` module. However, as mentioned previously, there are many instances where we would benefit from having access to type information in our graders. By design, grader code is only run on code that successfully typechecks. Therefore, we can make use of the compiler front-end to build up an instance of the compiler's internal `Typedtree` structure from the student's code AST, thereby gaining access to typing and scoping information for all identifiers.

However, the `Typedtree` structure is difficult to work with: libraries exist for easily pattern matching on and constructing `Parsetree` (untyped AST) fragments, but no such resources exist for working with the `Typedtree`. Additionally, during the process of constructing the typed AST, the untyped AST goes through a normalization process that is non-invertible: for instance, optional arguments to functions are filled in with their default values, labelled (keyword) arguments and record fields are sorted in alphabetical order, and

functions created using the `fun` construct are rewritten to use the `function` construct instead. Since we are specifically analyzing student code for features such as the use of certain code constructs, it is crucial for the representation of the student's code to be as close to its orignal form as possible. This is also very important for us to be able to provide students with clear and accurate error messages. Thus working with the `Typedtree` structure was not an option for us.

We have instead implemented our own augmented version of the OCaml parse tree, which we call `Typed_ast` to differentiate from the internal `Typedtree` structure of the OCaml compiler. The `Typed_ast` is structurally identical to the `Parsetree` structure: the difference is that some additional fields are added to some nodes in the tree to store scoping and typing information from the `Typedtree`. For instance, the type of expressions in the `Parsetree` includes the variant for identifiers `Pexp_ident of Longident.t loc`, while the corresponding variant in the `Typed_ast` is `Sexp_ident of Path.t * Longident.t loc`: the added field of type `Path.t` contains additional scoping information discovered during the typing process, including the identifier's full resolved module path and a unique stamp so that shadowed identifiers can be distinguished. Each expression node in the `Typed_ast` is also annotated with its inferred type, and the typing environment in which it was checked.

The `Typed_ast` is constructed from the `Parsetree` structure by first invoking the typechecking component of the OCaml compiler's front-end to construct a `Typedtree`; as mentioned previously, the grader is only run on code that typechecks, so this process always succeeds. We then traverse the untyped and typed ASTs in parallel via a series of mutually-recursive functions that handle different components of the AST: expressions, patterns, module expressions, etc. Since we have access to the untyped AST, we can map components of the `Typedtree` back to their original untyped forms, which allows us to "undo" the parts of AST normalization during the typing process that we previously described as non-invertible. The result is an instance of the `Typed_ast` structure corresponding to the student's code being graded.

Our `Typed_ast` structure covers only a subset of the OCaml language that we use in our course. In particular, it does not include imperative constructs such as `for` and `while` loops, nor any of OCaml's object-oriented features. It would not be difficult to extend the `Typed_ast` to add these constructs, but we are not currently interested in doing so.

Along with the data structure itself, our module provides several functions for working with the `Typed_ast`, many of which are similar to the ones provided by Learn-OCaml itself for working with untyped ASTs. We describe some of the most important ones below:

- `tast_of_parsetree_expression` and `parsetree_of_tast_expression` perform conversions between the `Typed_ast` and `Parsetree`, for interfacing with the functions from the grading libraries.

- `find_binding`, analogous to the function of the same name provided by the grading library for untyped ASTs, finds the top-level binding for a given name and retrieves the scoping information for that identifier, the `rec` flag, and the syntax tree for the binding expression.

- `ast_check_expr`, analogous to the function of the same name provided by the grading library for untyped ASTs, allows users to easily traverse a `Typed_ast` by specifying stateless functions that should be called on expressions, patterns, etc. without needing to implement any of the recursive traversal logic.

- `variables`, `bound_variables`, and `free_variables` calculate the sets of all variables, all bound variables, and all free variables, respectively, occurring within a given expression.

- Various helpers are provided for constructing `Typed_ast` fragments, which are particularly useful for performing comparisons with concrete expressions for functions such as `ast_check_expr`.

All of the extensions to the grading libraries that we will discuss in this chapter have been implemented atop our augmented `Typed_ast` structure, so that they are scope-safe: no extra effort is needed to take variable shadowing into account, since scoping information is included with all identifiers in the `Typed_ast`. Access to the additional typing and scoping information opens up a variety of new avenues for writing grader code aside from the ones described in this chapter. For instance, one can use it to check that a student is using higher-order functions, or that they are not using any functions involving the `unit` type (which would suggest that the student is trying to program imperatively). We believe that making this information available to instructors writing graders is an extremely important step in making it easier for instructors using Learn-OCaml to write powerful and expressive graders.

4.2 Checking tail recursion

Tail recursion is one of the first topics we cover in COMP 302; 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 4.1, 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
```

Figure 4.1: Exponentiation by repeated squaring

As described in Chapter 2.1, we also had issues with an exercise in Fall 2019 where we asked students to implement a function tail-recursively using continuations, since students' code is compiled to Javascript using the `js_of_ocaml` compiler, which only performs tail call optimization on some common tail call patterns. This meant that some tail-recursive implementations were still triggering stack overflows on large inputs when running the grader, making our dynamic verification method extremely unreliable.

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.

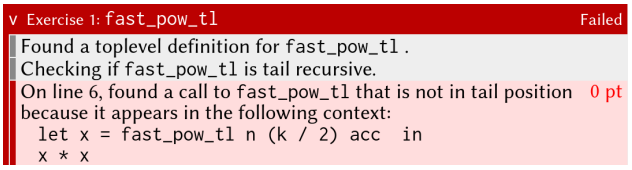
We have thus written our own syntactic check which attempts to determine if a function is tail recursive and incorporated it as part of a library in Learn-OCaml. The problem of determining whether or not a function is tail recursive is undecideable in the general case due to factors like aliasing and 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 most often come up in practice with student code. Given an identifier and a syntax tree, we recursively traverse the syntax tree for function calls to the identifier in question and check if any such function call occurs outside of tail position. If so, we flag it as an error and provide the student with the context in which the function call appears,

to explain why the given call is not a tail call. Since we perform our analysis over the `Typed_ast`, we do not explicitly need to keep track of any variable shadowing; we simply use the scoping information of identifiers in the typed tree to find all recursive calls to the desired function.

As an example, consider an attempt to implement exponentiation by repeated squaring tail-recursively using the helper function `fast_pow_tl` given in Figure 4.2a. 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, when given the `fast_pow_tl` identifier and the syntax tree of the function body, generates a report as shown in Figure 4.2b, 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



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

Figure 4.2: Tail recursion checking in Learn-OCaml

Our solution has a few drawbacks. The first is that our implementation needs to be given the identifier and syntax tree for the particular function to which all recursive calls should be tail calls. As this function is typically defined as an inner auxiliary helper, we need to do some extra work in our grader to search the student code for their helper function to obtain its name and code AST; and in the case of an implementation involving multiple mutually-recursive helper functions, we need to make sure that we are looking at all of them.

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

let rec pow_aux n k acc =
  pow n k

let pow_tl n k =
  pow_aux n k 1
```

Figure 4.3: An implementation erroneously accepted as "tail-recursive" by our grader

We opted instead to require students to define their helper functions at the top level with specific names, so that we could find them easily.

Another issue, which has come up multiple times in different forms in Chapter 2.1, is that students could define some other, non-local helper function that has the desired input/output behaviour but is not tail-recursive, and simply refer to this function within their implementation, as in Figure 4.3. In this example, the student is asked to implement a function `pow_tl` for performing exponentiation on non-negative integers, by writing a tail-recursive helper function `pow_aux`. The student instead defines another helper function `pow` that is not tail-recursive and uses this to implement `pow_aux`. No recursive call to `pow_aux` appears within its body; so, of course, our implementation does not find any recursive call to `pow_aux` that is not in tail position, and deems the student's code acceptable. We discuss the general idea of tracking non-local dependencies in Chapter 4.4.

We used our new library in the Fall 2019 term for one question on the first homework assignment in which students were asked to write a function to compute Fibonacci numbers tail-recursively. TAs manually screened students' submissions after the homework deadline and found that only a single submission out of over 300 implemented the function in a non-tail-recursive manner that was mistakenly accepted by the grader. The implementation used a trick similar to the one presented in Figure 4.3.

4.3 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. Michaelson [1996] proposes a linguistic characterization of functional programming style, by which style guidelines are categorized into four levels:

(1) The **lexical** level, which deals with the naming of identifiers.

(2) The **syntactic** level, which deals with issues of layout such as indentation, whitespace usage, and line length.

(3) The **semantic** level, which deals with the use of appropriate language constructs.

(4) The **pragmatic** level, which deals with documentation, such as comments, assertions, and type annotations.

The previously-named style issues are examples of style on the lexical and syntactic level. What we are particularly interested in, however, is 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
```

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, particularly since they are used to using boolean values to regular control flow constructs rather than as first-class values in their own regard. 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: we believe that continuously emphasizing ideas of good style to our students with examples involving their own code is of greater help to them than simply providing general rules.

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. Feedback from the style analyzer is given in two categories: *warnings* describe style issues where we feel that our rewritings should always be applied, while *suggestions* describe those more subjective issues of code style, where the student may or may not wish to apply the given rewriting.

Our extension contains several built-in style rules that can be enabled by default, as well as providing high-level functions for instructors to easily specify their own context-free rewrite rules, and lower-level functions for writing more complex checks that require context, such as counting the nesting depth of conditional expressions. We describe here the formal rewrite rules that we have implemented.

Simple boolean rewritings

```
e = true          ⟹  e

e = false         ⟹  not e

if e then true else false  ⟹  e

if e then false else true  ⟹  not e
```

These four rules deal with the unnecessary comparison of a boolean expression to `true` or `false`, and the unnecessary use of a branching conditional expression to construct a boolean value, as described previously. All four rules are categorized as warnings.

Boolean operators over conditional expressions

```
if e1 then true else e2  ⟹  e1 || e2

if e1 then e2 else false  ⟹  e1 && e2
```

These next two rules suggest using the operators `||` (boolean disjunction) and `&&` (boolean conjunction) in place of the equivalent conditional expressions. To avoid producing overly-complex and unreadable rewritings, these rules are only applied when the sub-expressions `e1` and `e2` are relatively "simple", or formally, when the syntax tree for each sub-expression has height at most 2. Both of these rules are categorized as suggestions; one might choose not to apply these transformations if the formulation as a conditional expression better expresses the intent of the code.

Single-clause pattern matchings as pattern-matching lets

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

After learning about pattern matching, the construct that tends to stick in our students' heads the most is, naturally, the `match` expression, since most of the examples we deal with in class involve pattern matching over datatypes with multiple constructors. Secondary to many students is the fact that function headers and `let`-expressions also allow for pattern

matching on values. We have thus introduced this rule, categorized as a warning, which reminds students that a `match` expression with only a single clause would be better written as a pattern-matching `let`.

Simple list rewritings

```
[e1] @ e2   ⟹   e1 :: e2
   e @ []   ⟹   e
   [] @ e   ⟹   e
```

These three rules deal with confusion about the `::` ("cons") and `@` (append) operators when dealing with lists. We find that many students struggle to correctly use the cons operator, which prepends an element to a list; often they will try to use it to concatenate two lists, or to append an element to the end of a list. Eventually some students will give up on using cons altogether, preferring to stick to the more familiar append operation for concatenating lists. They will thus write code putting an element `e1` into a singleton list so that a list `e2` can be appended to it, which is equivalent to using the cons operator. The first of these three checks will alert students that they may use the cons operator instead, though it is categorized as only a suggestion since we do not consider this a serious style issue. This check is not actually context-free: it will not be triggered if a singleton list is appended to another list as pair of a chain of appends, for instance as in the expression `e1 @ [e2] @ e3`.

The latter two rules deal with the appending of a list with an empty list (`[]`), which of course has no effect. While appending an empty list on the right (`e @ []`) does not cause much of a loss of efficiency, appending an empty list on the left (`[] @ e`) is an operation which takes time linear in the length of `e`. Both of these checks are categorized as warnings.

Pattern matching over explicit selectors

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

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

This rule (taken from Michaelson [1996]), which is categorized as a warning, emphasizes that pattern matching on the structure of a list is strictly preferable to comparing it to the empty list and using the explicit list selectors `List.hd` and `List.tl` to decompose it.

Eta reduction

```
fun pats1 pats2 ->
  e pats2
```
$\Longrightarrow$
```
fun pats1 -> e**
```

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

One concept that we have found students new to functional programming struggle to understand, and to use when appropriate, is that of *partial function application*: for instance, that the function `fun x -> 1 + x` could instead be written as `(+) 1`, or that the function `fun lst -> List.fold_left (+) 0 lst` (which sums up the elements of a list) could instead be written as `List.fold_left (+) 0`. This comes up most often when students are passing a function (typically one written as a previous part of the exercise in question) as argument to a higher-order function; very elegant solutions tend to be possible for these questions by using partial application. The process of factoring out the arguments to a function that do not need to be explicitly specified in this way is called *eta reduction*, and a function in which all arguments are explicitly specified is said to be fully *eta-expanded*. This rule is categorized as a suggestion, since one might sometimes prefer to write an eta-expanded version of a function for readability purposes.

Indeed, there are some cases in which performing eta reduction does not preserve the semantics of an expression. One example is when the type of an eta-reduced function is weakened by OCaml's *value restriction*, which is a restriction on polymorphism of expressions that are not syntactically values, such as function applications. A full description of the value restriction is outside the scope of this thesis; however, consider the definition of the following function `map_ident`, which maps the identity function onto a list:

```
let map_ident = List.map (fun x -> x)
```

One might expect `map_ident` to have the polymorphic type `'a list -> 'a list`, but the inferred type is instead `'_a list -> '_a list`, which is a *weakly polymorphic* type. Once `map_ident` is applied to a list, the type variable `'_a` is replaced by a concrete type, so the function is in fact monomorphic. However, if the definition of `map_ident` is eta-expanded, as follows:

```
let map_ident l = List.map (fun x -> x) l
```

then the type of `map_ident` is inferred to be polymorphic as expected. Thus applying an eta reduction here would in fact change the semantics of the function. This is one place where we can take advantage of the capabilities of the `Typed_ast`: since the style checker has access to the typing environment in which each expression is type-checked, and to the type-checking capabilities of the OCaml compiler, we can run the type-checker on our suggested rewritten expressions to ensure that the type of the expression has not changed after the eta reduction. If we find that our transformation has weakened the polymorphic type of a function, we will not suggest that eta reduction.

The other situation in which eta reduction can change the semantics of a function is when the function has side effects that occur before all arguments are supplied. For example, consider the following function, which, given an integer `init`, returns a "counter": a function that increments an internal counter (initialized to `init`) each time it is called, and returns the counter's new value.

```
let make_counter init =
  let counter = ref init in
  fun () -> incr init; init
```

The type of `make_counter` is `int -> unit -> int`, so one might think that the (rather contrived) function `fun () -> make_counter 0 ()` could be eta-reduced, without changing the semantics, to `make_counter 0`. However, the two expressions have different semantics: the

first function will return `1` each time it is called, while the second will increment an internal counter whenever it is called, returning a different value each time. Detecting such cases where eta reduction should not be applied is beyond the scope of our style checker, but thankfully these cases do not arise very often in our assignments.

For assignments where functions may have side effects that occur in this manner, we prefer to simply turn off the eta reduction check, since we can not determine for sure whether a function has side effects even with additional typing information from the `Typed_ast`. However, the presence of the `unit` type in a function's signature strongly implies that a function will have some sort of side effect, though we cannot determine when in the process of applying the function these effects may be triggered; thus one could consider modifying the eta reduction check so that it does not suggest partial applications of functions with the `unit` type in their signature.

Non-rewrite rules In addition to the rewrite rules listed previously, 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. This threshold is configurable by the instructor.

The implications of using the `Typed_ast` to implement our style checker, rather than the untyped `Parsetree` or OCaml's `Typedtree` structure, extend beyond its use in the eta-reduction check. Since the style checker applies rewrite rules to the student's code AST and prints code back to the student to suggest certain transformations, it is very important that the AST is as close in structure to the student's original code as possible; thus the `Typed_ast` was much better suited for this extension than the OCaml compiler's `Typedtree`. Some of our transformations, such as the ones on boolean expressions, introduce references to predefined identifiers (e.g. `&&`, `||`, `not`) that could potentially be overshadowed by another definition in the context in which the transformation occurs. Using the typing environments that annotate each expression node, we can check that any identifiers we are introducing have

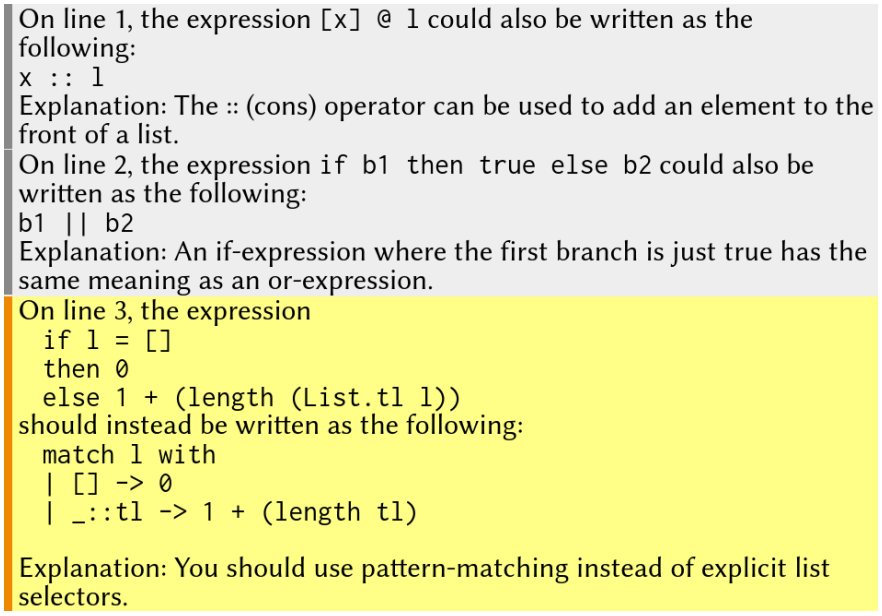


Figure 4.4: Sample feedback report from our style checker

not been overshadowed, and avoid suggesting their use if they have been. Thus, for instance, the style checker will not make any suggestions for the following expression:

```
let not x = false in
if b then false else true
```

even though, if `not` had not been overshadowed, it would have suggested rewriting the conditional expression as `not b`.

When the style checking extension is enabled in Learn-OCaml, a section on code style is added to the student's feedback report whenever they run the autograder on their code. A sample such report can be found in Figure 4.4, applying some of the previously-discussed rewritings. Note that the two categories of style feedback are differentiated in the report: warnings are those with a yellow background, while suggestions are those with a gray background. Applications of the rewrite rules can be accompanied by some optional further explanation, as specified by the instructor writing the checker.

Feedback reports are instances of a user-defined datatype in OCaml, so 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. When we deployed the style checking extension in the Fall 2019 term of COMP 302, we had the grader apply a blanket 1-point penalty to submissions triggering at least one style warning (yellow background only); we hypothesized that the desire to see a perfect score would incentivize students to pay attention to the style report and make corrections to their code. We report on how effective this was, as well as the usefulness of the style checker in general, in Chapter 5.

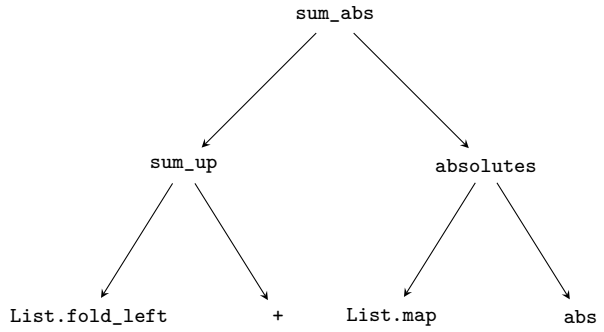
4.4 Dependency analysis

As has been discussed many times by this point, it became clear over our time using Learn-OCaml that we needed a way of transitively analyzing the functions that a student's implementation depends upon. When we require students to make use of a certain function or language feature, we need to be able to check any non-local helper functions the student may have written, rather than just looking at the body of the submitted function itself; and conversely, when we forbid students from using a particular function or construct, we need to make sure students aren't tricking the grader by hiding this usage within some other helper function that they then call within their implementation.

For instance, suppose we asked students to implement a function `sum_abs` which, given a list of integers, returns the sum of the absolute values of the items in the list, using the built-in functions `List.map` and `List.fold_left`. A student might implement `sum_abs` as in Figure 4.5a, by writing two helper functions: `absolutes`, which takes the absolute value of each item in a list (where `abs` is the integer absolute value function), and `sum_up`, which adds up the items in a list. If we were to grade this example by analyzing only the body of `sum_abs` itself, we would not find any usage of `List.map` or `List.fold_left`; however, the student has certainly made good use of those functions, and should be awarded full credit. On the other hand, if we wanted (for some reason) to *forbid* the usage of these list functions, the student would erroneously be given full marks, as the calls to `List.map` and

```
let absolutes lst =
  List.map abs lst
let sum_up lst =
  List.fold_left (+) 0 lst
let sum_abs lst =
  sum_up (absolutes lst)
```

(a) Implementation of a function to sum the absolute values of the items in a list



(b) Dependency graph corresponding to the implementation in (a)

Figure 4.5: Example of a dependency graph

`List.fold_left` are hidden within non-local helper functions.

To address this problem, we have extended the Learn-OCaml libraries with a function for generating a *dependency graph* from a given code AST. A dependency graph is a directed graph with vertices corresponding to the identifiers of predefined variables and the AST's top-level bindings. An edge exists from a vertex `v1` to a vertex `v2` if and only if the variable `v2` is used in the expression side of the binding of the variable `v1`. In this case, we say that `v1` directly depends on `v2`. If there exists a path in the dependency graph from a vertex `v1` to a vertex `v2`, we say that `v1` depends on `v2`. A dependency graph may contain cycles and even self-loops, expressing dependencies that arise from recursive definitions.

As an example, the dependency graph corresponding to the code for `sum_abs` above would have 7 vertices, connected as illustrated in Figure 4.5b. This graph expresses that `sum_abs` directly depends on `sum_up` (which, in turn, directly depends on `List.fold_left` and `+`) and `absolutes` (which, in turn, directly depends on `List.map` and `abs`). Thus `sum_abs` also transitively depends on the functions `List.fold_left`, `List.map`, `+`, and `abs`.

Given an AST, our implementation generates a dependency graph by traversing each top-level definition to determine the dependencies of the bound variables introduced by that definition. We do this by exploring the expression side of the binding and building an internal dependency graph for the variables (both local and non-local) that appear in the expression. Since we use the `Typed_ast`, no extra work is necessary to keep track of the shadowing

of variables; different variables with the same name are easily differentiated. We use this internal dependency graph to collect all of the free variables used in the binding expression; this also allows us to perform a basic unused variable analysis, since expressions with results that are discarded (as in, for example, `let _ = f x in ...`) do not contribute anything to the internal graph. Edges are then added to the graph between each bound variable introduced by the definition, and each free variable that appears in the expression side of the binding. Once we have a dependency graph, we can determine what free variables a particular function `f` depends upon by performing a breadth-first search starting from the vertex corresponding to `f` and keeping track of all visited vertices; the set of all visited vertices is precisely the set of `f`'s dependencies.

With access to the dependency graph for the student's code, we can track the usage of required or forbidden functions even when this occurs in non-local helper functions. We can also more thoroughly check whether a particular language feature has been used in an implementation, by analyzing not only the body of one particular function, but also the bodies of all of its (non-predefined) dependencies.

Our method for calculating the variables that should be counted as "used" by a particular function for grading purposes is neither sound nor complete. It is not sound because it is still possible to trick the grader into believing a variable has been used when it has not: for instance by using OCaml's `ignore` function, or creating a function like it, which simply discards its argument. It is not complete because if an expression is evaluated only for its side effects, with the results being discarded, the variables in that expression will not be considered as having been "used".

We considered these deficiencies in our dependency graph method acceptable for our purposes, and made use of this extension for writing graders in the Fall 2019 term. As described in Chapter 2.2, we manually screened the auto-graded submissions after the assignment deadlines passed. In an exercise on higher-order list functions, where students were instructed to implement functions on matrices using particular higher-order functions, we found that 4

students had tricked the grader by inserting an unnecessary usage of a required higher-order function into their code, out of nearly 300 submissions. We received no complaints that the grader was deducting marks for not using a required function when the function had, in fact, been used.

Very recently, an idea quite similar to our method of dependency checking has been implemented as a tool for automatically grading Haskell programs [de Vries 2019].

Chapter 5

Evaluation

In the Fall 2019 offering of COMP 302, we made use of the extensions to the Learn-OCaml platform described in Chapters 3 and 4 to design and write graders for the 6 homework assignments given that term. We invited students taking the course to enroll in a study to help us improve the platform by allowing us access to the data from all of their submissions to the autograder for each assignment over the course of the term. After removing students who either dropped the course, alternated between different Learn-OCaml accounts to work on their assignments (which would make their progress difficult to trace), or attempted 3 or fewer of the homework assignments, we were left with 71 students participating in our study out of the 299 enrolled in the course. We also administered a survey at the end of the term on students' experiences using the Learn-OCaml platform, to which 201 out of 299 students responded.

In this chapter, we present visualizations of the data we gathered from students' homework submissions, along with our observations. Note that a rigorous scientific study judging the effectiveness of our extensions to the platform is outside the scope of this thesis; such a study would need to be carefully planned in advance and span multiple years, collecting data from students working on assignments with and without our extensions, in a more controlled environment with confounding factors sufficiently addressed. We therefore do not claim to

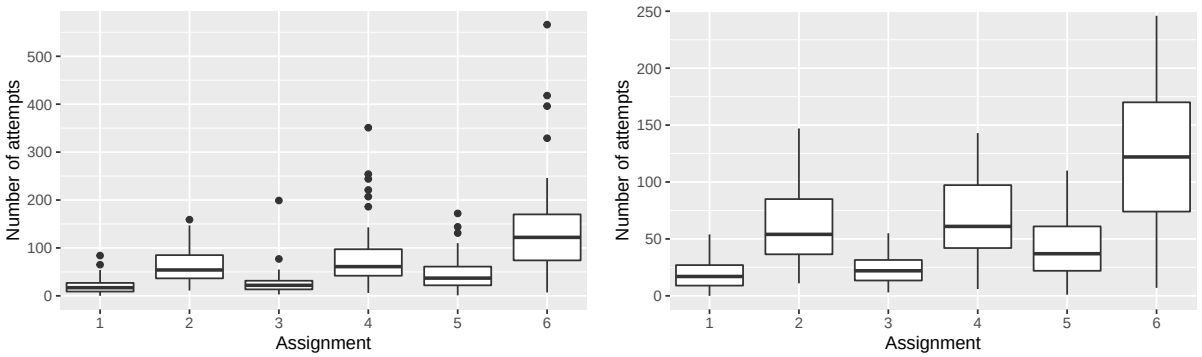
be able to come to any solid conclusions based on our data, but instead point out interesting trends that occur in our data and speculate on the possible reasons behind them. We also discuss the responses to our survey on using the platform, and compare them to the results from a very similar survey filled out by students in the Fall 2018 term, before we deployed any of our extensions to Learn-OCaml.

5.1 Mining data from homework submissions

Each account ("token") created on the Learn-OCaml platform is given its own Git repository. A new commit is made to this repository whenever the student modifies their account (e.g. changing their nickname), manually syncs their work, or runs the autograder of a currently active assignment. The data stored in this commit includes the student's submission, the grade it was given by the autograder, and the automatically-generated feedback report for the student's code. The feedback report, stored as a JSON object, provides us with a wealth of information: using our knowledge of the structure of sub-reports generated by our extensions, we can traverse the feedback report to gather useful statistics such as the number of style suggestions and warnings, the implementation and mutation testing scores on specific questions, and even the identifiers of the exact mutants detected by the student's submission. Thus, by going through their tokens' commit histories, we can trace students' progress as they work on their assignments and get a picture of how they are using the Learn-OCaml platform, and our extensions in particular. We have done this for the 71 students who consented for their data to be used in our study. In this section we present some visualizations of the data we collected.

5.1.1 Number of grading attempts

Over the course of the term, our students were given 6 homework assignments, each covering some core functional programming concepts:



(a) (b)

Figure 5.1: Distribution of the number of times students ran the auto-grader for each assignment. Figure (a) displays the full range of data including outliers, while Figure (b) displays the data with outliers removed, but still considered for statistical calculations.

- **Homework 1**: Basic OCaml; simple recursion and tail recursion
- **Homework 2**: Pattern matching and recursion on lists; user-defined datatypes
- **Homework 3**: Using higher-order functions
- **Homework 4**: Writing higher-order functions; references and state
- **Homework 5**: Modules; control flow using exceptions and continuations
- **Homework 6**: Programming language implementation

Figure 5.1 shows the distribution of the number of times each student successfully ran the auto-grader for each homework assignment. A "successful" run of the grader is one in which the grader is able to produce a full grading report; a grader run may be unsuccessful if the student attempts to grade code that does not typecheck, if the student aborts the grader before it is finished running (which usually means the student's code has entered an infinite loop), or if an internal error occurs in the browser's JavaScript engine while executing the student's code. We observe that the median number of attempts was noticeably higher on assignments 2, 4, and 6, with medians of 54, 61, and 122 attempts respectively; assignment 5 then had a median of 37 attempts; while assignments 1 and 3 took most students a

relatively low number of attempts, with medians of 17 and 22, respectively. There are a couple of factors which may account for some of this difference:

(1) Assignments 2, 4, and 6 each contained at least one question that students particularly seemed to have trouble with, as evidenced by questions asked on the discussion boards:

- On assignment 2, students had to write tests for and implement a function over propositional formulas (defined using a user-defined datatype) which would rewrite a formula by recursively applying a set of rewrite rules. Students had particular difficulty thinking of the class of test case in which multiple rewrite rules needed to be applied.
- On assignment 4, students needed to implement a function `remove` for a doubly-linked list defined using references as pointers without duplicating any data cells in memory, which required a strong understanding of how their linked structures were represented in memory.
- Assignment 6 required students to implement programming language concepts such as type-checking and evaluation for a toy language. The topic of programming language fundamentals is generally new to all of our students at this stage and students have historically considered the assignment on this to be the most challenging.

(2) We note also that Assignments 1, 2, and 6 were the ones that focused the most on having students write their own test cases, although since it was the first homework, Assignment 1 was relatively easy. Assignment 4 required students to write some test cases that they were able to come up with fairly easily, while we did not consider writing tests for most of the functions in Assignments 3 and 5 to be very instructive (see Chapter 3.3). When students found themselves stuck trying to think of test cases to expose a particular mutant, they would end up running the grader several times

trying out test cases until they found one that worked. This may partially explain the lower number of attempts for Assignments 3 and 5.

5.1.2 Use of the mutation testing functionality

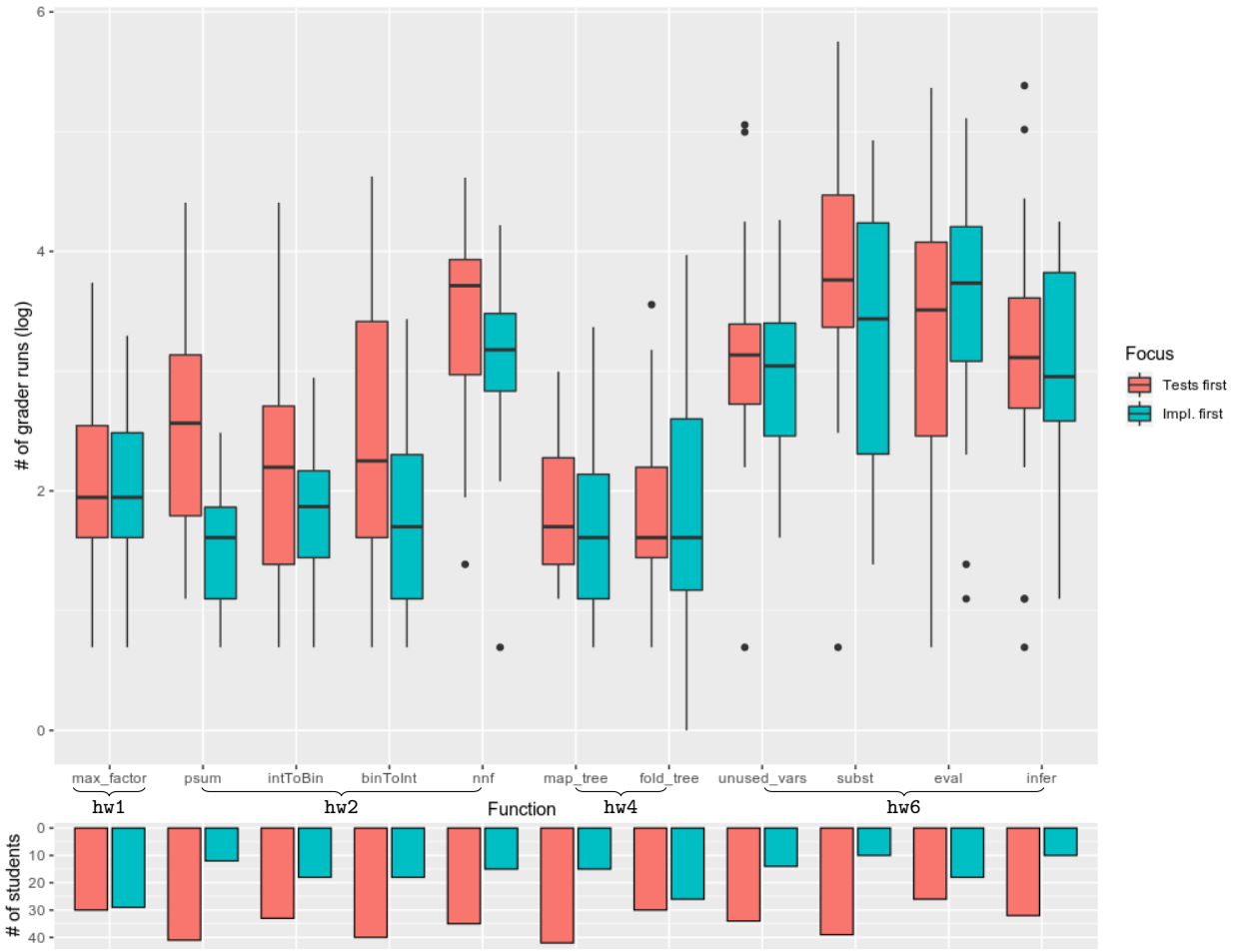


Figure 5.2: Distribution of the (approximate) number of grading attempts for functions for which students were required to submit both test cases and an implementation. Students are grouped based on whether they appeared to start by writing tests or by writing an implementation. The histogram shows how many students fell into each category for each function.

To visualize how students were using the mutation testing functionality, and how effective it was at encouraging test-driven development, we roughly categorized students into two groups for each homework question where they were required to submit both test cases and

an implementation: *test-focused* and *implementation-focused*. Students were considered to be following a *test-focused* strategy for a question if either: a) they obtained a score of 100% on their tests on their first grading attempt for that question, but did *not* score 100% on their implementation on that same attempt; or b) the difference between their normalized test and implementation scores was non-negative and non-decreasing over their first two grading attempts for that question, and positive on their second attempt. Intuitively, this criteria means that the student's scores on their test cases were increasing at a faster rate than their grade on their implementation. Conversely, students were considered to be following an *implementation-focused* strategy for a question if either: a) they obtained a score of 100% on their implementation on their first grading attempt but did *not* score 100% on their tests on that same attempt; or b) the difference between their normalized test and implementation scores was non-positive and non-increasing over their first two grading attempts for that question, and negative on their second attempt. Students that did not meet either criterion for a particular question were not sorted into a category for that question.

Since we did not specify a threshold for the difference between a student's test and implementation scores for that student to be considered test-focused or implementation-focused, and these criteria are in general not very rigorous, this is only a very rough basis for categorization. In addition, looking at a student's test and implementation scores gives us only part of the picture; it would be more informative to look at the submitted code itself to see how much the test cases and function implementation were being changed between grading attempts, though this would come with its own challenges of tracing and checking the code of any helper functions as well (similar to the discussion in Chapter 4.4). As our data collection scripts were not written in OCaml, having them inter-operate with our dependency analysis functions would require significant effort, and would be better suited for a future study. One particularly illustrative example of how relying just on scores can give a skewed picture was a question on the first homework assignment in which students were asked to write tests for and then implement a function `fib_tl` which calculated Fibonacci

numbers tail-recursively. Student solutions were given 0 by the grader if they were not tail-recursive; thus even if a student was attempting the implementation first, this would not show in their scores if they did not achieve a tail-recursive solution on their first try. Since the classifications were likely to be inaccurate for this question, we have omitted it from our results.

The upper graph in Figure 5.2 shows the distribution of the approximate number of grading attempts for each function where students were required to submit both test cases and an implementation (besides `fib_tl`, as mentioned above), for students that were considered *test-focused* versus those that were considered *implementation-focused*. The lower histogram displays the number of students that fell into each category for each function. The number of grading attempts a student made for a particular function was calculated by counting the number of grading attempts from the first time the student received a non-zero score on either their tests or implementation, until the time the student achieved their maximum combined test and implementation score for that question. Note that this does not include any later attempts at running the grader without an increase in score if the student does not reach 100% and continues trying; and if a student alternates between working on multiple questions on an assignments, these numbers would be inflated. As before, a better approach in the future would be to look at which functions are being modified in the code itself.

The histogram shows that more students appeared to follow a test-first approach than an implementation-first approach for each of the functions displayed, indicating that we may have successfully encouraged students to at least start working on questions by writing a test case instead of jumping straight into the implementation. However, following a test-first approach does not appear to decrease the number of grading attempts a student made when compared to those who followed an implementation-first approach; in fact, the data appears to suggest that students following a test-first approach ran the grader more times on average than those who focused on the implementation first. It is possible that this is due to the larger number of students following a test-first approach for most of the questions, leading to

a larger sample size and more possibility for variation compared to the students following an implementation-first approach. For the two questions that had a relatively small difference between the number of students following each approach (`max_factor` and `fold_tree`), the median number of grader runs between the two categories seems quite similar; for a third function, `eval`, which had a difference of fewer than 10 students between the two categories, the median number of grader runs is a bit smaller for students who followed a test-first approach compared to an implementation-first approach. Meanwhile, all of the functions with a difference of 20 or more students between the test-first and implementation-first categories (`psum`, `binToInt`, `nnf`, `map_tree`, `unused_vars`, `subst`, and `infer`) all had a higher median number of grader runs from the students in the test-first category versus the implementation-first category. It would therefore be interesting to see more data on this over the next several runs of the course to try to determine if a judgement can be made on how the approach a student follows affects the number of times they run the grader.

Another, more informative, direction would be to have some students implement these functions without being instructed to write tests or being given access to any of the mutation testing functionality, while other students would be required to submit test cases and be graded on them before being allowed to start on their implementation. We could then compare the number of grading attempts each group of students makes before achieving their maximum score on their implementation and see if there is a difference between the students that wrote test cases and those that did not, without some of the confounding factors in our study as discussed above.

5.1.3 Use of the style checker

As discussed in Chapter 4.3, our style checking extension provided students with a style report along with their grading report each time they ran the grader, which gave them suggestions on how to rewrite their code to achieve better functional programming style. We have collected data on students' style reports on each of their grading attempts for the

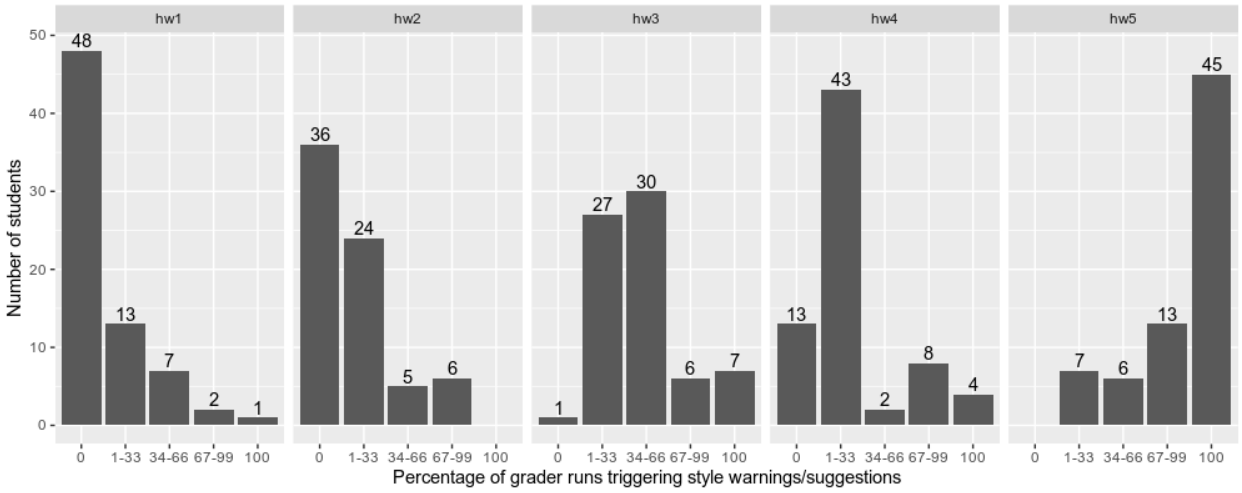


Figure 5.3

first five homework assignments (the style checker was mistakenly not enabled on the final assignment).

For each student on each assignment, we calculated the percentage of grading attempts where students triggered at least one style suggestion or warning from the style checker. The distribution of these percentages for each homework assignment can be seen in Figure 5.3. We have graphed students who did not trigger any style suggestions or warnings on any of their grading attempts (0% column) and those who triggered at least one suggestion or warning on every single one of their grading attempts (100% column) in their own columns as we find these two categories to be of specific importance. Students in the 0% category are those who never made any style errors that would be detected by our style checking extension on a particular assignment, while those in the 100% category are those who either never attempted to follow the suggestions given by the style checker, or were unsuccessful in doing so.

It was quite a surprise to see that 48 out of 71 students in the study triggered no style suggestions or warnings at all on the first assignment, since this was their first experience with writing their own OCaml code. One of the questions on the first assignment involved writing a function to compute the largest factor of a positive integer `n` smaller than `n` itself;

while students would likely be able to easily understand how to implement this using a loop in the imperative languages that they had previous experience with, the challenge with this question was thinking about the process in a recursive manner. Another question required students to write a function to compute fibonacci numbers tail-recursively, which posed similar challenges. We expected most students to stumble upon style warnings or suggestions while experimenting to figure out how to approach these questions. It is very possible that students were making other sorts of style mistakes that our style checker does not detect at the moment; it could be valuable future work to look at students' submissions manually to come up with further ways to extend our style checker. Overall the results for the first homework assignment seem quite positive, with the majority of students triggering style suggestions or warnings at most one-third of the time, which suggests that students were taking feedback from the style checker into account and fixing their style mistakes without too much delay. The second homework assignment also appears similar in this regard.

The third homework assignment is where the results begin to shift. The number of students who did not trigger any style suggestions or warnings at all drops to only 1. We believe this is likely because the third homework assignment focused on using higher-order functions, and students needed to provide partially-applied versions of their own functions as arguments to functions like `List.map` and `List.fold_left`. We have found that partial application is a concept that students have historically had trouble applying consistently to their work, so it is quite likely that a lot of students initially tried to use eta-expanded versions of their functions as arguments to higher-order functions, triggering eta-reduction suggestions from the style checker. The relatively low number of students whose submissions triggered style suggestions or warnings more than two-thirds of the time suggests that most students were taking the feedback from the style checker into account and partially-applying their functions accordingly.

The most dramatic change in distribution occurs in the results for the fifth homework assignment, where the majority of students triggered style suggestions or warnings for 100%

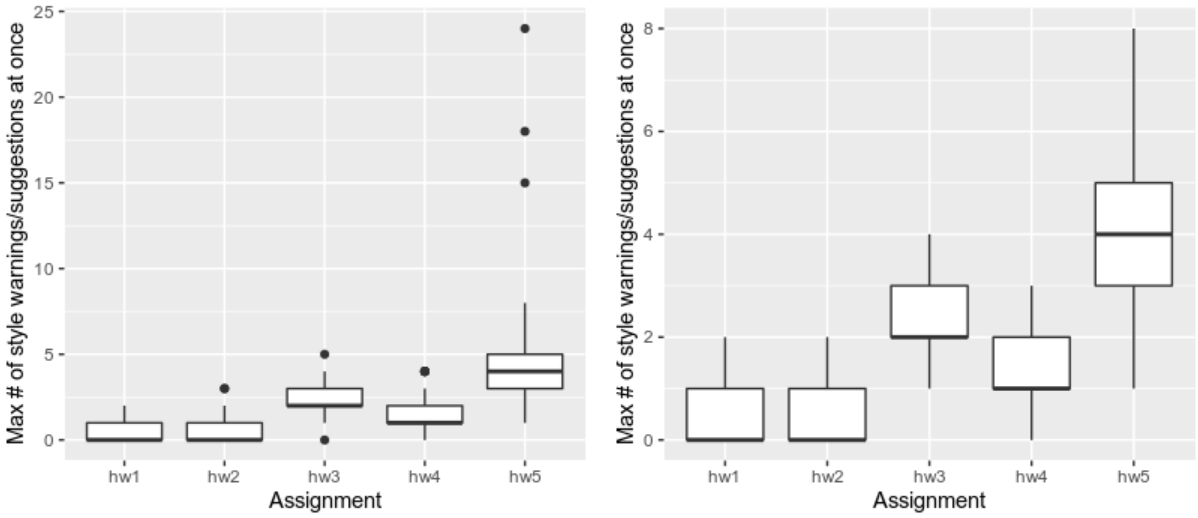
of their submissions. In the fifth assignment, students were required to write a module implementing operations on a "metric", represented internally using floating-point numbers and operations (see **Modules** in Chapter 2.1). The signature for this module included functions for adding two values of the metric's internal type `t` as well as multiplying by a scalar floating-point number, which were implemented simply by using OCaml's built-in floating-point operations as follows:

```
let plus x y = x +. y
let prod s x = s *. x
```

As the given template code included the stubs `let plus x y = raise NotImplemented` and `let prod s x = raise NotImplemented`, students naturally just filled in the function bodies and then ran the grader, resulting in the style checker suggesting that these definitions be eta-reduced as follows:

```
let plus= (+.)
let prod = ( *. )
```

We believe that most students simply ignored these suggestions because they did not understand them. While students are expected to be more familiar with the concept of partial application at this point of the course, it is our experience that students find it difficult to let go of the notion that built-in operators such as `+` and `-` are special constructs of the language (as is true in the languages they are used to, such as Python and Java), rather than functions like any others with some special syntax for convenience. The requirement that these built-in operators need to be surrounded by parentheses to be treated as names likely adds to this confusion. In addition, we find that students tend to be reluctant to modify the headers of function stubs given in the template code, believing that it may invalidate their submissions in the eyes of the grader. We believe it is very likely that the significant shift in distribution on this assignment is due to students not acting on the style checker's suggestions to eta-reduce these functions. On one hand, this is somewhat encouraging as it indicates that students are not simply blindly copying style suggestions that they do not

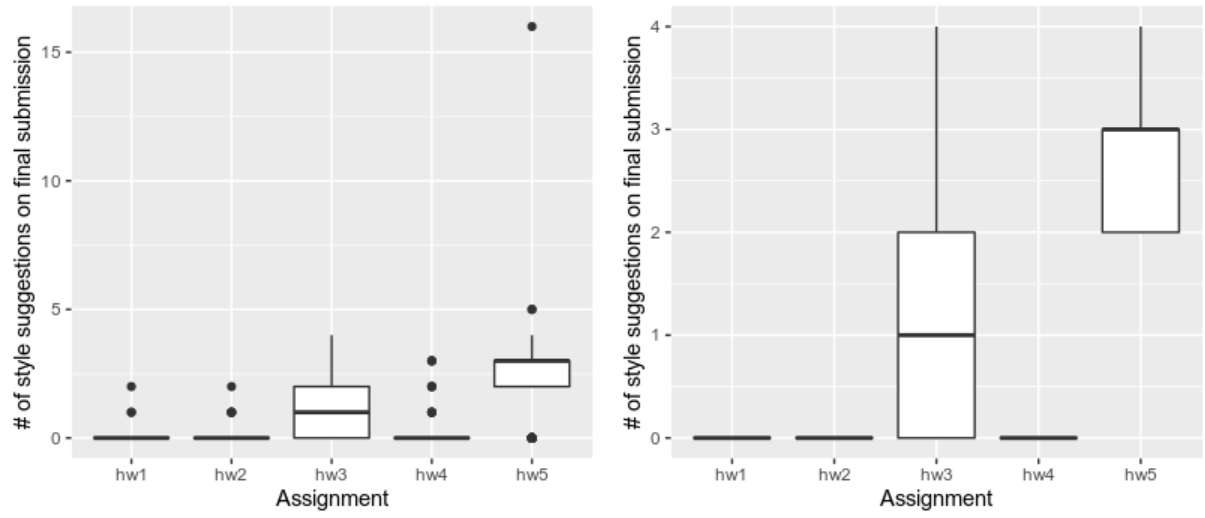


(a) (b)

Figure 5.4: Distribution of the maximum number of style warnings/suggestions students triggered at once for each assignment. Figure (a) displays the full range of data including outliers, while Figure (b) displays the data with outliers removed, but still considered for statistical calculations.

understand. On the other hand, we would like students to be able to learn from the style suggestions they are given. It might be helpful to explicitly include reading these style suggestions, trying to understand why they are being given, and asking questions if unsure, as an ungraded part of this specific assignment to encourage more students to ask about style suggestions they do not understand.

Figure 5.4 shows the distribution of the maximum number of style warnings/suggestions each student triggered for a single submission, for each assignment. This data serves to complement the observations made from the previous graph: the median maximum number of style warnings triggered at once on the first assignment was 0, and the the third and fifth assignments had noticeably higher medians. On the fifth assignment, the median value is 4 style warnings/suggestions at one time, while the first and third quartiles were 3 and 5 style warnings/suggestions at once, respectively. As discussed above, we expect that the cause of many students' style warnings/suggestions were eta-reduction suggestions pertaining to simple functions students needed to implement within a module operating on floating-point



(a) (b)

Figure 5.5: Distribution of the number of style suggestions triggered by students' final submissions for each assignment. Figure (a) displays the full range of data including outliers, while Figure (b) displays the data with outliers removed, but still considered for statistical calculations.

numbers; there were 3 such functions (`plus`, `prod`, and `toString`), so this data fits within an expected model where students triggered these 3 eta-reduction suggestions, ignored them, and proceeded to trigger 0-2 more style warnings in a single grader run over the course of the assignment.

Figure 5.5 shows the distribution of the number of style *suggestions* (not including style *warnings*) triggered by each student's final submission for each assignment. Our graders for each assignment deducted a single point from the student's score if the style checker produced any style warnings (as opposed to suggestions, as described in Chapter 4.3), in order to encourage students to fix the style corrections that we found particularly important. As a result of this, only 3 final submissions (out of 354) over the entire semester triggered a style warning from the grader; the rest triggered only style suggestions, or no warnings/suggestions from the style checker at all. We have thus graphed only the number of style suggestions reported for students' final submissions here. The data shows that the majority of students addressed all of their style warnings by their final submissions for assignments 1, 2, and 4,

with the students who did not do so being outliers. As with the previous graphs, we see noticeably different results for assignments 3 and 5. We suspect that these two assignments are different because they were the two assignments where students were likely to trigger eta-reduction suggestions from the style checker; assignment 3 was about using higher-order functions, while assignment 5 involved implementing a simple module as we have described previously. This suggests that we should focus extra attention on making the concept of eta-reduction clear to our students.

One thing that is noticeably missing from our data is definite statistics on what kinds of style warnings/suggestions students were triggering, which would allow us to make some stronger conclusions rather than simply suspecting the major cause of many students' style warnings. For the purpose of future studies, it would be very useful to add some sort of "error code" to the warnings/suggestions given by the style checker, so that we can easily determine automatically which classes of warnings and suggestions are present in a grading report and make use of this when visualizing our data.

We have also graphed students' grade trajectories for each assignment against the number of style warnings they triggered on each grader run to see how these two factors interacted. To protect the privacy of our students, we cannot publish these graphs; however, we can share observations we have made from looking at them. The most interesting thing that we noticed with some students was a pattern of students' grades decreasing at the same time as their number of style warnings/suggestions decreased, suggesting that some students were trying to follow the advice given by the style checker, but mistakenly changing the logic of their code in the process. We would then sometimes see the student's grade return to its previous value on the next grader run, with the style warning/suggestion re-introduced and remaining for quite some time, suggesting that students would then give up on trying to correct style errors since they could not figure out how to do so without changing the behaviour of their code. In Figure 5.6 we have tried to summarize how often students were able to successfully follow guidance from the style checker versus how often they failed

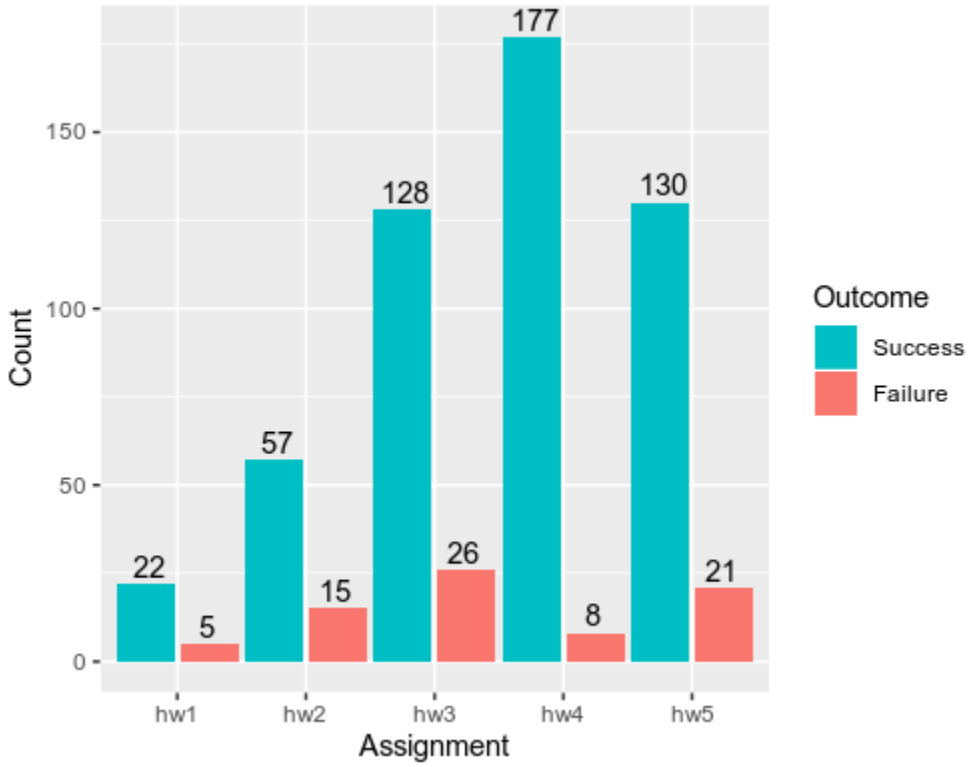


Figure 5.6: Outcomes of students' attempts to address style warnings/suggestions given by the grader on each assignment. "Successful" attempts were those that fixed the flagged code without a decrease in grade, while "failed" attempts were those where the code was corrected, but the student's overall assignment grade decreased in the process.

to do so. We have approximated these situations from our data as follows: a student was considered to have "successfully" fixed a style warning/suggestion if, between two consecutive runs of the grader, their number of style warnings/suggestions decreased, but their grade did not decrease. A student was considered to have "failed" to fix a style warning/suggestion if, between two consecutive runs of the grader, the number of style warnings/suggestions decreased, but their grade decreased as well. Since this does not take into account that a student may have fixed a style warning/suggestion but introduced another at the same time, or that a student may have written large amounts of code between grader runs so that an increase/decrease in grade does not necessarily indicate success or failure at addressing a style warning/suggestion, this is not a fully accurate measure of how successful students were at incorporating feedback from the style checker. However, since we found that a lot

of students tended to make small changes to their code aimed at fixing one thing at a time, we think that this measure is still useful. Figure 5.6 indicates that failures to fix style errors were relatively uncommon. Once again, it would be useful to collect data on what kinds of style errors were associated with successful and unsuccessful corrections, to determine what particular classes of style suggestions/warnings students had trouble addressing.

5.2 End-of-term student survey

At the end of both the Fall 2018 and Fall 2019 terms of COMP 302, we asked students to complete a short survey on their experiences with using Learn-OCaml for their homework assignments during the term. The survey responses were anonymous to all teaching staff and all questions on the survey were optional. Students were offered a 1% grade increase for the entire class if 75% of enrolled students (relaxed to around 67% in Fall 2019 due to a late survey release) submitted a response, to incentivize our students to complete the survey. In Fall 2018 we received responses from 275 out of 347 enrolled students, and in Fall 2019 we received responses from 200 out of 299 enrolled students. In this section we discuss the relevant survey results, including a comparison between responses in Fall 2018 and Fall 2019 for shared questions, and some additional questions added in Fall 2019 related to our mutation testing and style checking extensions.

As discussed in Chapter 2.2, we found that students were not making as much use of the toplevel as we would have liked in the Fall 2018 term, as evidenced by the results of the survey. We addressed this in the Fall 2019 term by demonstrating the usage of the Learn-OCaml toplevel regularly in lectures. Figure 5.7 shows the responses to a survey question "How often did you use the OCaml toplevel to test your assignment code?" from students in the Fall 2018 and Fall 2019 offerings of COMP 302. Compared to the Fall 2018 responses, the responses from Fall 2019 see weight shift from students who answered that they used the OCaml toplevel "never", "rarely", or "occasionally", to answering that they used the

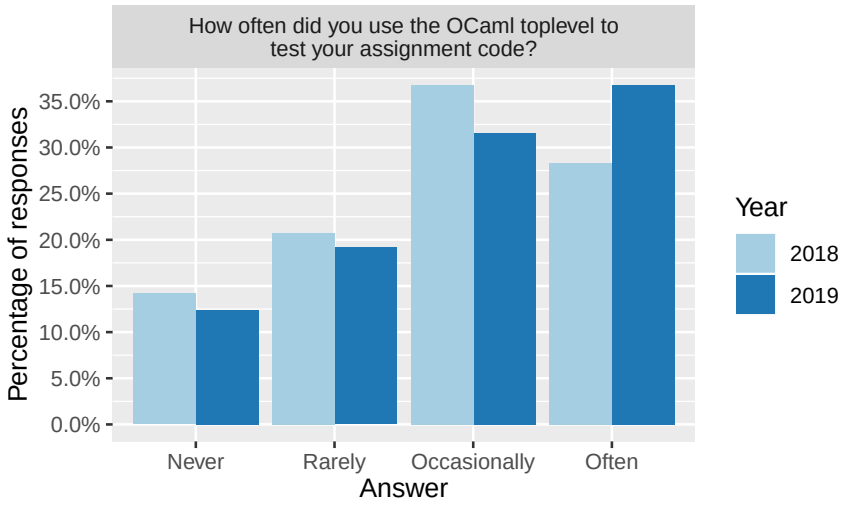


Figure 5.7: Responses to a survey question on how often students used the OCaml toplevel to test their own assignment code, asked in both Fall 2018 and Fall 2019. The light blue bars correspond to the Fall 2018 students' answers, while the dark blue bars correspond to the Fall 2019 students' answers. Response counts are graphed as percentages rather than raw numbers for the purpose of comparison.

toplevel "often". In total the percentage of students who answered "Often" increased from around 28.36% in Fall 2018 to around 36.79% in Fall 2019, which is a promising sign that our changes made to emphasize use of the toplevel in Learn-OCaml were effective.

There was also a set of more general questions asked in both Fall 2018 and Fall 2019 pertaining to the use of auto-grading in COMP 302. As shown in Figure 5.8, students were asked to rate how much they agreed or disagreed with a series of statements relating to using Learn-OCaml. Ratings were given on a Likert scale, translated to numbers for our graphs with 1 meaning "strongly disagree" and 5 meaning "strongly agree". One observation to be made when comparing numbers between Fall 2018 and Fall 2019 is that students tended to agree more that the error messages from the grader were clear and helpful in Fall 2019 as compared to Fall 2018; this implies that individual adjustments we made to our graders as described in Chapter 2.1, for instance the custom printing of doubly-linked lists, were generally well-received. On the other three graphs, we find that the percentage of students responding positively (answers 4-5) seems roughly similar across the two course offerings. Observing the bottom two graphs in the figure, which indicate how positively students felt

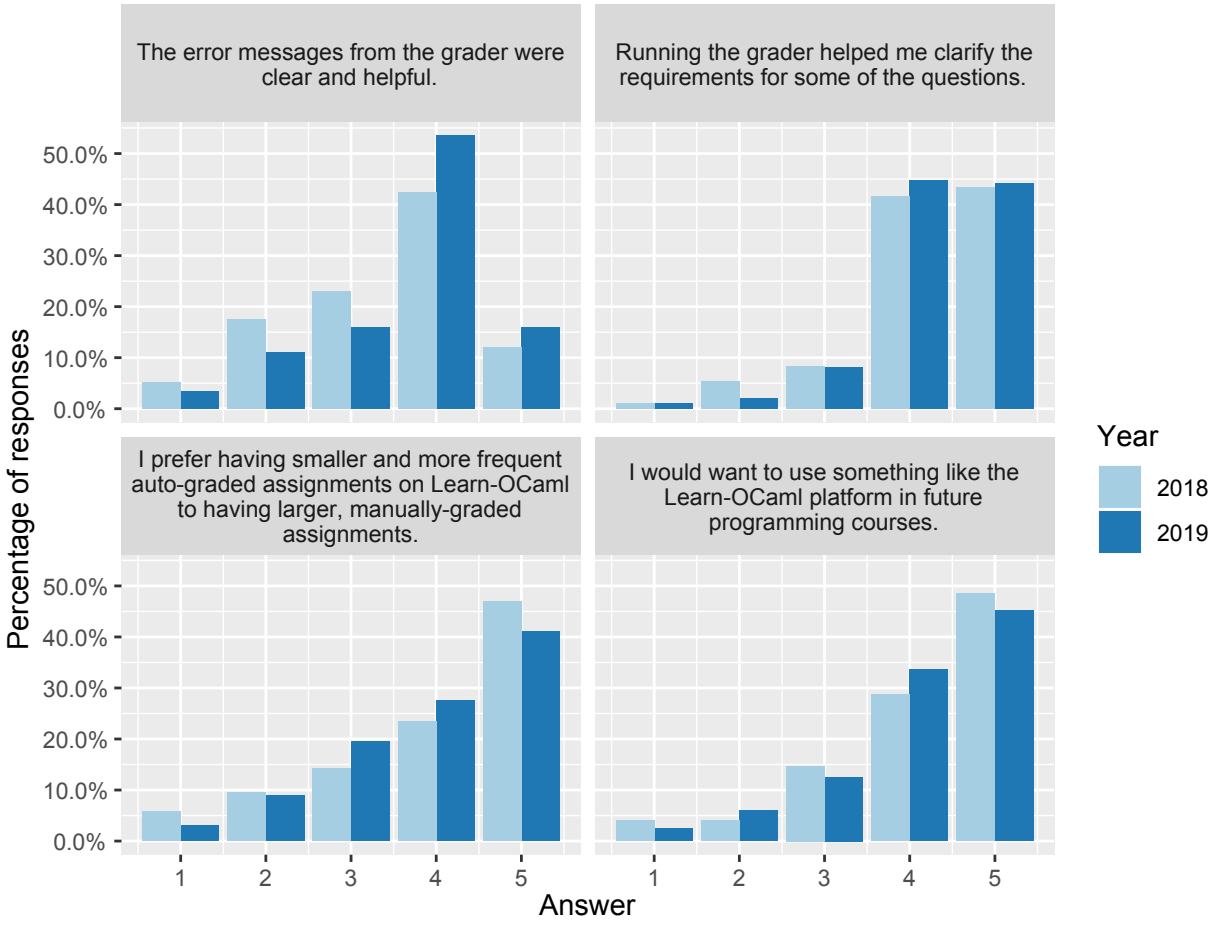


Figure 5.8: Responses to survey questions relating to students' general experiences with auto-grading, asked in both Fall 2018 and Fall 2019. Students were were asked to indicate how much they agreed or disagreed with each statement on a Likert scale; these answers are translated to numbers on the graphs where 1 corresponds to "strongly disagree" and 5 corresponds to "strongly agree".

about using the Learn-OCaml platform in COMP 302, we notice that students tended less towards extreme opinions ("strongly disagree" or "strongly agree") in Fall 2019 as compared to Fall 2018.

In the Fall 2019 survey we also asked some questions to determine how students felt about the mutation testing functionality. The responses to these questions are shown in Figure 5.9. As before, students were asked to indicate how much they agreed or disagreed with a series of statements on a Likert scale, translated to numbers for our graphs with 1 meaning "strongly disagree" and 5 meaning "strongly agree". The large number of students

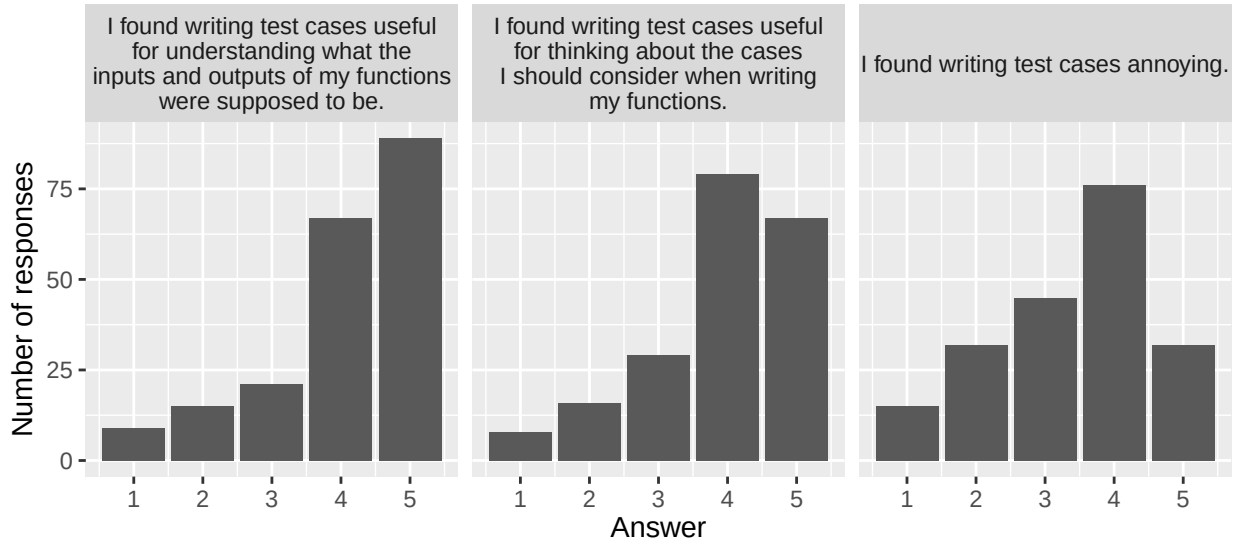
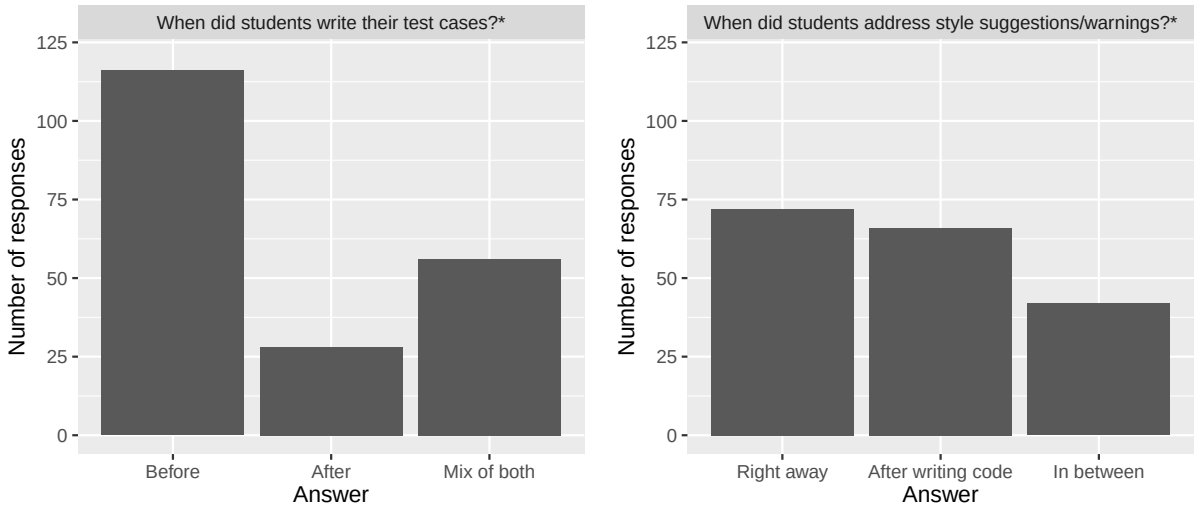


Figure 5.9: Responess to survey questions relating to the mutation testing extension introduced in Fall 2019. Students were asked to indicate how much they agreed or disagreed with each statement on a Likert scale; these answers are translated to numbers on the graphs where 1 corresponds to "strongly disagree" and 5 corresponds to "strongly agree".

answering that they found writing test cases to be annoying is hardly a surprise. However, there was also a generally positive response to the other two statements, pertaining to how useful students found it to write test cases using the mutation testing extension. This indicates that students did understand and appreciate the value in writing test cases before writing their code despite finding the practice annoying, which we consider to be a win.

Finally, we asked a couple of questions to determine when students were writing their test cases, and when students were addressing style warnings/suggestions as provided by the style checker. Responses to these questions are graphed in Figure 5.10. The exact text of the questions and response options are not shown on the graph, but instead shortened to save horizontal space.

As graphed in Figure 5.10a, we asked students, "Did you usually write your test cases before or after you implemented the function they were supposed to be testing?"? Students were given the answer options "Before", "After", "An even mix of both", and "Not sure / don't remember"; only 1 student answered "Not sure", which has been omitted from the graph. The majority of students answered that they wrote their test cases before imple-



(a) (b)

Figure 5.10: Responses to survey questions relating to how students used the mutation testing (a) and style checking (b) extensions introduced in Fall 2019. Questions and response options on the graphs do not exactly correspond to the questions and response options that students were shown on the survey, but have been shortened to save horizontal space.

menting their functions, with about a quarter answering that they did an even mix of both, which roughly agrees with our earlier findings from examining the number of students in our survey who appeared to be taking a "tests-first" versus "implementation-first" approach.

As graphed in Figure 5.10b, we asked students, "Which of the following **best** describes how you used the feedback from the style checker (the style reports at the bottom of the grading feedback)?". Students were given the following answer options: "I fixed style issues as soon as they came up" (graphed as "Right away"), "I put off fixing style issues until I was done writing my code" (graphed as "After writing code"), "Somewhere in between" (graphed as "In between"), and "Not sure / don't remember" (21 responses, omitted). Answers to this question were more evenly-distributed, with similar number of students answering that they fixed their style suggestions/warnings right away and that they waited until they were finished writing all of their code.

Chapter 6

Conclusion

In Fall 2018, we began using the Learn-OCaml online programming platform to deliver automatically-graded homework assignments in COMP 302, the introductory undergraduate course on programming languages and paradigms at McGill University. In this thesis we have given a detailed account of our experience in using Learn-OCaml out of the box as instructors new to using automated grading. We have described the concepts tested in these online homework assignments, and the issues we encountered as students became over-reliant on the grader's automated feedback and began focusing on the correctness of their code above all else, as this was the only criteria the grader judged them by. This motivated us to write extensions for Learn-OCaml to 1) encourage students to test their own code by grading students on how many bugs their test suites were able to catch; and 2) make it easier for us to write graders that provide automated feedback on properties of code besides simple input/output correctness, such as functional programming style. We have reported on how we used these extensions in the Fall 2019 offering of COMP 302 to better pursue these teaching goals, and on how these extensions were received by our students. Finally, we have provided data gained from a study of the grader submissions made by a subset of our students and observed that our extensions appear to have been somewhat effective in encouraging desired testing behaviours in our students; we have followed this up with a

discussion on changes that could be made to improve this result further in future offerings of our course.

It is our hope that the experiences shared in this thesis, along with our shared repository of homework problems and extensions to the Learn-OCaml platform, will make it easier for other instructors unfamiliar with automated grading to offer high-quality functional programming courses to a wider audience, so that we can promote best functional programming practices for students everywhere.

6.1 Future work

The work we have done on the Learn-OCaml platform is just a start. There still remains much that can be done to make Learn-OCaml a more effective tool for teaching and learning functional programming, in the directions of improving usability for students, making it easier for instructors to write exercises, and simplifying the use of the platform for semi-automated grading, to name a few. In the future, there are several projects that we would like to see come to fruition; some are currently being worked on by undergraduate research students, while others require a larger commitment.

Interactive lessons As briefly mentioned in Chapter 1, in addition to the exercise functionality, the Learn-OCaml platform allows instructors to create interactive tutorials for their students. These tutorials function as a sort of interactive textbook, including paragraphs of text interspersed with code snippets that students can click to send to the toplevel to be evaluated. Compared to the exercises, which are the main feature of Learn-OCaml, the tutorial component of the platform is quite underdeveloped. We would like to see this functionality expanded, and the interface made more user-friendly, so that we can convert our course notes to this format to help facilitate independent learning.

Student-friendly error messages The error messages raised by the OCaml type-checker are a common source of confusion for our students, due to the type-checker's left-to-right bias when inferring types resulting in errors that they may not expect. For instance, when inferring the type for a conditional expression `if b then e1 else e2`, the type-checker will infer a type for the expression `e1` and then try to unify this with the type of `e2`. If unification fails, the type-checker will always flag the expression `e2` as causing the error, even though the issue may in fact stem from the expression `e1`. Researchers have taken multiple approaches to this problem, such as suggesting modifications to the type inference algorithm [Charguéraud 2015] and creating systems for interactive type debugging [Tsushima and Asai 2011; Chitil 2001]. It would be extremely useful to explore how these could be integrated into the Learn-OCaml system.

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 and evaluate partial programs, as in the live programming environment Hazel Omar et al. [2017, 2019]. 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. Additionally, the dynamic semantics that Hazel incorporates for typed holes allows evaluation to proceed around these holes, and even around ill-typed expressions, making it easier for students to debug their programs and understand the compile-time and run-time errors they may experience.

Incremental stepping Somewhat related to the above, the DrRacket environment includes an algebraic stepper[1] 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

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

of the Caml language. Expanding this to a larger subset of the language, and implementing this in OCaml so that it can be integrated with the Learn-OCaml system, would allow students to explore how different expressions are evaluated and demystify this process for them.

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.

Semi-automated grading Learn-OCaml features a rudimentary system for instructors to view and browse student submissions for their homework assignments, but it is quite underdeveloped and not very convenient to use. We would like to either flesh out this functionality so that teaching assistants can more easily navigate students' submissions for a particular exercise, or integrate the grading reports produced by Learn-OCaml with some external system for course management. The latter seems a particularly good direction to explore, so that the focus of the Learn-OCaml platform can remain on delivering exercises and tutorials rather than being split on course management utilities.

Bibliography

Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research*, 3(1):245–262, 2004.

Tiffany Barnes and John Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In *International Conference on Intelligent Tutoring Systems*, pages 373–382. Springer, 2008.

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

R E. Berry and B A.E. Meekings. A style analysis of C programs. *Commun. ACM*, 28(1): 80–88, January 1985. ISSN 0001-0782. doi: 10.1145/2465.2469. URL `http://doi.acm.org/10.1145/2465.2469`.

Tracy Camp, W. Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. Generation CS: The growth of computer science. *ACM Inroads*, 8(2):44–50, May 2017. ISSN 2153-2184. doi: 10.1145/3084362. URL `http://doi.acm.org/10.1145/3084362`.

Benjamin Canou, Grégoire Henry, Çagdas Bozman, and Fabrice Le Fessant. Learn OCaml, an online learning center for OCaml. In *OCaml Users and Developers Workshop 2016*, 2016.

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

Arthur Charguéraud. Improving type error messages in OCaml. *arXiv preprint arXiv:1512.01897*, 2015.

Peter M Chen. An automated feedback system for computer organization projects. *IEEE Transactions on Education*, 47(2):232–240, May 2004. ISSN 0018-9359. doi: 10.1109/te.2004.825220. URL `http://dx.doi.org/10.1109/te.2004.825220`.

Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ACM SIGPLAN Notices*, volume 36, pages 193–204. ACM, 2001.

Rohan Roy Choudhury, Hezheng Yin, and Armando Fox. Scale-driven automatic hint generation for coding style. In *International Conference on Intelligent Tutoring Systems*, pages 122–132. Springer, 2016.

Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449–452. ACM, 2016.

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

Charlie Daly. RoboProf and an introductory computer programming course. *ACM SIGCSE Bulletin*, 31(3):155–158, 1999.

Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

R.H. de Vries. Walker: Automated assessment of Haskell code using syntax tree analysis, June 2019. URL http://essay.utwente.nl/78785/.

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

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

Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5 (3):4, 2005.

Stephen H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '03*, page nil, - 2003a. doi: 10.1145/949344.949390. URL https://doi.org/10.1145/949344.949390.

Stephen H Edwards. Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In *Proceedings of the international conference on education and information systems: technologies and applications EISTA*, volume 3, 2003b.

Stephen H. Edwards and Manuel A. Pérez-Quiñones. Experiences using test-driven development with an automated grader. *J. Comput. Sci. Coll.*, 22(3):44–50, January 2007. ISSN 1937-4771. URL http://dl.acm.org/citation.cfm?id=1181849.1181855.

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

Mattias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The DrScheme project: an overview. *ACM Sigplan Notices*, 33(6):17–23, 1998.

Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 369–388. Springer, 1997.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(2):159–182, 2002.

Ronald A Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*, pages 26–27. Oliver and Boyd Ltd, London, 3rd edition, 1948.

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

Vincent Gramoli, Michael Charleston, Bryn Jeffries, Irena Koprinska, Martin McGrane, Alex Radu, Anastasios Viglas, and Kalina Yacef. Mining autograding data in computer science education. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '16, pages 1:1–1:10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4042-7. doi: 10.1145/2843043.2843070. URL `http://doi.acm.org/10.1145/2843043.2843070`.

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

Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, and Athanasios Tsintsifas. The CourseMarker CBA system: Improvements over Ceilidh. *Education and Information Technologies*, 8(3):287–304, 2003.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource aware ML. *Lecture Notes in Computer Science*, page 781–786, 2012. ISSN 1611-3349. doi: 10.1007/978-3-642-31424-7_64. URL http://dx.doi.org/10.1007/978-3-642-31424-7_64.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. *ACM SIGPLAN Notices*, 52(1):359–373, Jan 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009842. URL http://dx.doi.org/10.1145/3093333.3009842.

Jack Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529, 1960.

David Jackson. A semi-automated approach to online assessment. *ACM SIGCSE Bulletin*, 32(3):164–167, 2000.

David Jackson and Michelle Usher. Grading student programs using ASSYST. *ACM SIGCSE Bulletin*, 29(1):335–339, 1997. doi: 10.1145/268085.268210. URL https://doi.org/10.1145/268085.268210.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, Sep. 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.62.

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

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. MuCheck: An extensible tool for mutation testing of Haskell programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 429–432. ACM, 2014.

Nguyen-Thinh Le, Sven Strickroth, Sebastian Gross, and Niels Pinkwart. A review of ai-supported tutoring approaches for learning programming. In *Advanced Computational Methods for Knowledge Engineering*, pages 267–279. Springer, 2013.

Michael Luck and Mike Joy. A secure on-line submission system. *Software: Practice and Experience*, 29(8):721–740, 1999.

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

Paul W Oman and Curtis R Cook. A taxonomy for programming style. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 244–250. ACM, 1990.

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

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages*, 3(POPL): 14, 2019.

Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. An empirical study of iterative improvement in programming assignments. In *46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*, pages 410–415. ACM, 2015. doi: 10.1145/2676723.2677279. URL `https://doi.org/10.1145/2676723.2677279`.

Joe Gibbs Politz, Daniel Patterson, Shriram Krishnamurthi, and Kathi Fisler. Captainteach: Multi-stage, in-flow peer review for programming assignments. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*, pages 267–272. ACM, 2014.

Kenneth A Reek. The TRY system-or-how to avoid testing student programs. In *ACM SIGCSE Bulletin*, volume 21, pages 112–116. ACM, 1989.

Michael J Rees. Automatic assessment aids for Pascal programs. *ACM Sigplan Notices*, 17 (10):33–42, 1982.

Konstantinos Sagonas and Thanassis Avgerinos. Automatic refactoring of Erlang programs. *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming - PPDP '09*, 2009. doi: 10.1145/1599410.1599414. URL `http://dx.doi.org/10.1145/1599410.1599414`.

Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. *ACM SIGCSE Bulletin*, 33(3):133–136, 2001. doi: 10.1145/507758.377666. URL `https://doi.org/10.1145/507758.377666`.

Mark Sherman, Sarita Bassil, Derrell Lipman, Nat Tuck, and Fred Martin. Impact of auto-grading on an introductory computing course. *J. Comput. Sci. Coll.*, 28(6):69–75, June 2013. ISSN 1937-4771. URL `http://dl.acm.org/citation.cfm?id=2460156.2460171`.

Zach Sims and Ryan Bubinski. Codecademy. 2011. URL `http://www.codecademy.com`.

Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. Experiences with Marmoset: designing and using an advanced submission and testing system for programming courses. In *11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*, pages 13–17, 2006. doi: 10.1145/1140123.1140131. URL `https://doi.org/10.1145/1140123.1140131`.

Guy Tremblay and Éric Labonté. Semi-automatic marking of Java programs using JUnit. In *International conference on education and information systems: technologies and applications (EISTA'03)*, pages 42–47, 2003.

Kanae Tsushima and Kenichi Asai. Report on an OCaml type debugger. In *ACM SIGPLAN Workshop on ML*, volume 3, 2011.

Jérôme Vouillon and Vincent Balat. From bytecode to JavaScript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.

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

John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. Who tests the testers? In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 51–59. ACM, 2018.

Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. WeScheme: the browser is your programming environment. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 163–167. ACM, 2011.