Engaging, Large-Scale Functional Programming Education in Physical and Virtual Space

Kevin Kappelmann Jonas Rädle Lukas Stevens
Fakultät für Informatik
Technische Universität München, Germany
kevin.kappelmann@tum.de raedle@in.tum.de stevensl@in.tum.de

Worldwide, computer science departments have experienced a dramatic increase in the number of student enrolments. Moreover, the ongoing COVID-19 pandemic requires institutions to radically replace the traditional way of on-site teaching, moving interaction from physical to virtual space. We report on our strategies and experience tackling these issues as part of a Haskell-based functional programming and verification course, accommodating over 2000 students in the course of two semesters. Among other things, we fostered engagement with weekly programming competitions and creative homework projects, workshops with industry partners, and collaborative pair-programming tutorials. To offer such an extensive programme to hundreds of students, we automated feedback for programming as well as inductive proof exercises. We explain and share our tools and exercises so that they can be reused by other educators.

1 Introduction

This paper reports on strategies and solutions employed to run two iterations of a large-scale functional programming and verification course at the Technical University of Munich (TUM). While the first iteration (winter semester 2019, 1057 participants) took place on campus, the second iteration (winter semester 2020, 1031 participants) was affected by the COVID-19 pandemic and took place in virtual space. Previous iterations of the course were introduced in [2]; however, we were facing two novel challenges:

Soaring Enrolments The relatively young field of computer science has become one of the largest study programmes around the globe. The increase of student enrolments is dramatic [12, 11] while employment of new teaching staff often lags behind. At TUM, the number of new enrolments in computer science more than doubled between 2013 and 2021 from 1110 to 2644 (an increase of 138%) while academic staff only increased from 439 to 573 (31%) [1].

This drastic increase not only requires more physical resources – like larger lecture halls and more library spaces – but also academic staff for supervision. Given the discrepancy in growth between student enrolments and staff employment, automation of supervision and feedback mechanisms is inevitable. Automation, however, should not negatively affect the quality of the teaching.

Online Teaching The ongoing COVID-19 pandemic forced a radical transition from on-site teaching to online classes. Lecturers had to rethink the way they present material and interact with students, teaching assistants the way they assist students in tutorial sessions. Students, on the other hand, suffer from a lack of social interaction and communication, leading to higher levels of stress, anxiety, loneliness, and symptoms of depression [6].

In our experience, the disconnect between students and lecturers, as well as the lack of on-campus interaction between students may also lead to *cramming*: the practice of showing little participation during

Submitted to:
TFPIE 2022

the semester while studying extensively just before the exam. Cramming tends to result in poor long-term retention and shallow understanding of material. Indeed, the benefit of spacing learning events apart rather than cramming has been demonstrated in hundreds of experiments [8, 5].

Besides these two general challenges, there is a third – subject-specific – challenge we were keen to tackle:

Functional Programming is Practical Feedback by students and personal experience has shown us that many students at TUM question the applicability and usefulness of functional languages beyond academia. They are disappointed by a lack of industrial insight and real world – or at least interactive – applications. Indeed, some even perceive functional programming as an obstacle; after all, they already know how to program imperatively.

Good educators do not just teach but inspire: we have to bring the benefits of functional languages closer to our students' hearts by showing real-world applicability and making functional programming fun and engaging.

In this paper, we present our answers to these challenges and provide tools and exercises for other educators to make functional programming lectures – in physical or virtual space – engaging and scalable. Our resources can be found in the central repository of this paper [1].

Outline We begin by describing the general framework and underlying conditions of the course and its syllabus in Section 2. In Section 3, we describe the tools and teaching methods that we used during lectures. Section 4 describes the mechanisms, tools, and technical setup we used to create an engaging experience that is scalable for the practical part of the course, including tutorials, homework assignments, and competition systems. Section 5 introduces "Check Your Proof" – a tool created by our lab that automatically checks simple inductive proofs for Haskell programs. Section 6 describes how we adapted our exams to the COVID-19 situation and the large number of participants, and Section 7 concludes with a summary and aspects to improve in future.

2 Course Structure and Conditions

2.1 Conditions

The 5 ECTS[2] course was mandatory for computer science undergraduates in their third semester and an elective for other related degrees such as games engineering or information systems. All students studied Java in their first semester and had taken courses on algorithms and data structures, discrete mathematics, and linear algebra. The course ran for 14 weeks with one 90-minute lecture, one 90-minute tutorial, and one exercise sheet each week.

1057 students registered for the first iteration[3] that took place on campus in winter semester 2019 (WS19) and 1031 for the second iteration[4] in winter semester 2020 (WS20), taking place in virtual space due to the COVID-19 pandemic.

[1] https://github.com/kappelmann/engaging-large-scale-functional-programming/
[2] European Credit Transfer System; one ECTS credit equals 30 hours of work
[3] https://www21.in.tum.de/teaching/fpv/WS19/ (website – except "Wettbewerb" – German; course material English)
[4] https://www21.in.tum.de/teaching/fpv/WS20/ (English)

Both iterations were organised by the lecturer, Tobias Nipkow, and the authors of this paper. The former designed the course, created the slides[5], and delivered the lectures. The others took care of the practical and organisational part of the course. All gained valuable experience in running an online course on the theory of computation for 1071 students in summer semester 2020. Finally, Manuel Eberl had the honour of assisting us with the weekly programming competition in his role as the *Master of Competition* (see Section 4.1).

Needless to say, running tutorials for more than 1000 students each semester on our own is impossible. We were further assisted by 13 student assistants in WS19 and 22 student assistants in WS20. In WS19, their primary job was to run the tutorials and provide feedback for homework submissions (e.g. code quality). However, it became clear to us that this manual feedback is not effective and that their time is better spent creating engaging exercises (see Section 4.1).

2.2 Syllabus

The course deals with the basics of functional programming and the verification of functional programs. Most parts of the course could be done using any functional language. We chose Haskell because of its simple syntax, large user community, and good testing facilities (in particular QuickCheck). The syllabus of the course stayed close to the one presented in [2]. The changes are the omission of the parser case study (parsing problems were instead introduced in homework exercises), the rigorous introduction of computation induction and type inference, and the decision to split off I/O from monads and introduce it earlier in the lecture. The last is done in an effort to show the students that the most important side effects they know from imperative languages can be recovered in pure functional languages. For ease of reference, we list the syllabus below. New or modified topics are marked ($*$):

1. Introduction to functional programming
2. Basic Haskell: `Bool`, QuickCheck, `Integer` and `Int`, guarded equations, recursion on numbers `Char`, `String`, tuples
3. Lists: list comprehension, polymorphism, a glimpse of the Prelude, basic typeclasses (`Num`, `Eq`, `Ord`), pattern matching, recursion on lists (including accumulating parameters and non-primitive recursion), scoping by example
4. Proof by structural induction and computation induction on lists ($*$)
5. Type inference algorithm ($*$)
6. Higher-order functions: `map`, `filter`, `foldr`, λ-abstractions, extensionality, currying, more Prelude
7. Typeclasses
8. Algebraic datatypes and structural induction
9. Concrete I/O without introducing monads ($*$)
10. Modules: module syntax, data abstraction, correctness proofs
11. Case studies: Huffman codings and skew heaps
12. Lazy evaluation and infinite lists
13. Complexity and optimisation

[5] https://www21.in.tum.de/teaching/fpv/WS20/assets/slides.pdf

14. Monads (∗)

Concepts are introduced in small, self-contained steps. Characteristic features of functional programming languages such as higher-order functions and algebraic data types are only introduced midway through the course. This makes the design of interesting practical tasks harder but ensures that students are not overwhelmed by the diversity of new principles that are not part of introductory imperative programming courses. In general, the course progresses from ideas close to what is known from imperative languages (e.g. boolean conditions, recursion on numbers, auxiliary functions, etc.) to simple applications of new concepts (e.g. recursion and induction on lists) to generalised new concepts (e.g. algebraic data types and structural induction).

3 Lectures

We used a mix of slides, live coding, and whiteboard proofs for the lectures. Each self-contained topic came with small case studies and examples. The latter were accompanied by suitable QuickCheck tests and inductive correctness proofs when appropriate. The proofs presented during the lectures stayed close to the format accepted by the proof checker that was used in the practical part of the course (see Section 5). The live coding sections in particular received positive feedback by the students.

In both iterations, students were allowed to interact and ask questions at any time. However, synchronous interaction with hundreds of students is challenging: While the lecturer cannot answer all questions due to time constraints, many students are also too reserved to ask questions given the large audience. As the semester progresses, interaction tends to degrade to questions posed by a small community of motivated students; questions shared by a majority, on the other hand, often go unheard.

In WS19, we partly addressed this problem by offering an asynchronous Q&A forum[6] where students could post anonymously or using their real name. The forum contained separate sections for the theoretical part (including the lectures) and the practical part of the course. Questions posted in the former were answered by the lecturer to increase interaction between students and the lecturer – the lack of which was often criticised by students in our department. Questions in the latter were also answered by teaching assistants and by other students. Answers by students were explicitly encouraged by us and outstanding contributions awarded a special prize at the end of the semester.

However, while the forum was a great success with over 3800 posts per semester, engagement in the theoretical section stayed far behind its practical counterpart ($\leq 2\%$). Moreover, the forum does not fully address the live interaction problem since questions are answered asynchronously. As such, students stuck with conceptional problems may not be able to keep up with the rest of the lecture, leading to frustration.

For the second iteration of the course – the online semester – we thus added a new interaction method. The lectures were livestreamed and interaction was made possible by means of a live Q&A board[7]. The board was moderated by a PhD student sitting in the same room as the lecturer. Questions could be answered and voted on by students as well as the moderator. The moderator approved and answered individual and simple questions directly while forwarding questions of general interest to the lecturer in order to increase engagement between the students and the lecturer.

We can report that this moderated format increased engagement when compared to the first iteration: students were less reluctant to submit questions because 1) they had the chance to ask questions anonymously, 2) they were not afraid to "interrupt" the lecture, and 3) they were able to ask new kinds of

[6] We used a Zulip instance hosted by our department: https://zulip.com/case-studies/tum/

[7] We used tweedback https://tweedback.de/

questions. Examples of the third include discussion of alternative solutions by students, organisational questions, and slightly off-topic discussions that nevertheless increase engagement and curiosity. We recommend to offer a moderated live Q&A board even for lectures taking place on campus or running in a hybrid format.

Following the livestream, the recordings were uploaded for asynchronous consumption. Students watching the lectures asynchronously still had the chance to submit questions to the forum and to receive answers by the lecturer.

4 Practical Part

4.1 Engagement Mechanisms

Students spend the majority of their time on the practical part of the course. This is where they apply the theory explained in the lecture to tutorial and homework exercises in the form of programming tasks, proof exercises, and miscellaneous other assignments (type inference, transformation of programs into tail-recursive form, etc.). As each student has unique interests, strengths and weaknesses, and different levels of commitment, we aim to employ a diverse set of mechanisms to keep them engaged.

As outlined in the introduction, keeping up student engagement is particularly challenging in courses that are taught remotely. We experienced this first hand when we taught a course in theoretical computer science during the first semester affected by the COVID-19 pandemic. We saw a significantly larger decrease in homework and tutorial participation over the course of that semester than in previous iterations. We thus attempted to put a particular emphasis on engaging teaching methods for the functional programming course in WS20.

We want to emphasise that engagement does not simply increase by offering more things – this may even increase stress – but by offering things that serve neglected needs. Good engagement mechanisms not only keep students busy but genuinely make the course more fun. Participation should originate from curiosity and enjoyment rather than pressure.

Here we describe the engagement mechanisms that were particularly valuable for our course:

Grade Bonus The most straightforward of all mechanisms is that of the grade bonus. For both iterations, students were able to obtain a bonus of one grade step on their final exam provided that they achieved certain goals in the practical part of the course. This mechanism was already used in a previous iteration but subsequently dropped because of negative experiences with plagiarism. However, as a result, participation in homework exercises severely decreased [2]. Moreover, the student council reported to us that one of the most asked for wishes by students is that of a grade bonus.

We hence re-introduced the bonus with some changes. First, instead of asking for 40% of all achievable points, we changed to a pass-or-fail per exercise sheet system. Students passed a sheet if they passed $\approx 70\%$ of all tests and obtained the bonus if they passed $\approx 70\%$ of all sheets. We changed to this system so that students could not obtain the grade bonus early on in the semester and then stop participating, leading to cramming.

Secondly, in WS20, we introduced additional ways to obtain bonus points, for example by participating in programming contests or workshops by industry partners. This diversified the system and offered new ways to obtain the bonus, in particular for those students that were struggling with programming tasks but were nevertheless interested in the course material. As a result, out of 802 students that interacted with the homework system at all, 298 students obtained the grade bonus in WS20 (37%).

In contrast to previous years, we have not seen any severe cases of plagiarism despite running all submissions through a plagiarism checking tool[8].

Instant Feedback An observation we already made in Section 3 extends to the practical part of the course: feedback must come fast. Again, an asynchronous Q&A forum helps in this regard, at least for questions of a general nature. Questions and problems specific to the submission of a student (e.g. a bug or error in a proof), however, must be fixed by the student themselves as 1) it is a critical skill of any computer scientists to discover bugs and 2) code/proofs may not be shared before the submission deadline due to the grade bonus.

Automated tests can fill this gap: they provide direct feedback (e.g. failing input and expected output pairs) with little delay without giving away too much information. Needless to say, they are also an important mechanism to scale the homework system to a large number of students. We describe our testing infrastructure in more detail in Section 4.2.

However, we still let student assistants manually review all final submissions in the first iteration of the course. We specifically instructed them to provide feedback not covered by automation, in particular regarding code quality. To our dismay, we have to report that this feedback did very little and most of it was probably ignored. In part, this is because it took 1–2 weeks after each submission deadline to provide feedback to all students. At that point, the students had already moved on to a fresh set of exercises and were probably not motivated to revisit their old submissions. Some students may also only care about passing the tests and are thus not particularly interested in feedback about code quality.

In our second iteration, we hence reallocated resources: instead of grading submissions, student assistants now supported us by creating engaging exercises and offering new content (e.g. running workshops with industry partners) while we focused on writing exhaustive tests with good feedback and extended our automated proof checking facilities (see Section 5). To provide at least some feedback on code quality, we instructed students to use a linter (see Section 4.2).

We can report very positively on this reallocation: we were able to offer a more diverse set of exercises and had the resources to offer new content while quality of code did not seem to suffer. Indeed, the linter even seemed to increase students' awareness to not only write correct code but also use good coding patterns. This seems to be due to the fact that 1) the linter provides instant feedback and 2) it visually highlights affected code fragments and provides quick fixes.

Competition and Awards Due to positive feedback, we continued the tradition of running an opt-in weekly programming competition as introduced in [2]. Each week, one of the homework assignments was selected as a competition problem and a criterion for ranking the correct submissions was fixed. The range of problems and ranking criteria were diverse, including code golf challenges, optimisation problems, game strategy competitions, an ACM-ICPC-like programming contest, and creative tasks like music composition and computer art. We introduce some of the exercises and share their templates in Section 4.3. The top 30 entries received points and were presented to the public on a blog[9], written using the ironic self-important third-person style established in previous semesters. The top 30 students then received awards at a humorously organised award ceremony at the end of the semester.

To improve on previous iterations, we made participation more exciting by organising better awards. To avoid fierce competition – after all, we do not want to host a fight for money – we did not announce the

[8] We used Moss https://theory.stanford.edu/~aiken/moss/

[9] https://www21.in.tum.de/teaching/fpv/WS20/wettbewerb.html (WS20) and https://www21.in.tum.de/teaching/fpv/WS19/wettbewerb.html (WS19)

value of the awards nor did we announce any allocation key until the very end of the semester. However, we added some hints on our blog to make students curious. We cooperated with industry partners to offer prizes such as tickets to functional programming conferences, Haskell workshops and programming books, as well as cash and material prizes. This initial contact with industry partners also sparked the idea to offer real-world Haskell workshops run by software engineers from industry in WS20 (explained further below).

The competition in WS20 also greatly benefited from incorporating the work of our student assistants: At the beginning of the semester, we brainstormed for competition ideas. We then formed teams, each one being responsible for the implementation of one idea to be published as a competition exercise during the semester. This allowed us to create more extensive, diverse, and practical exercises than in previous iterations, where all tasks were created by the course organisers.

As reported in [2], we can confirm that the competition works extremely well to motivate talented students. They go well beyond what is taught as part of the course when coming up with competitive solutions. Many of them continue to become major drivers in the team of student assistants in follow-up iterations. Indeed, after offering the competition in WS19, we received more applications for student assistant positions in WS20 (they more than doubled) than we could hire – something we had never experienced before. We also received testimonies from students that even though they did not perform well or did not participate at all in the competition, they nevertheless enjoyed the humorous blog posts and advanced material discussed on it.

Despite the work of our student assistants, however, running the competition remains an enormously labour-intensive task, in particular the evaluation of submissions and the writing of the blog post. We envisage further assistance by student assistants in future iterations in those regards.

Workshops with Industry Partners Many students at TUM have questioned the applicability and value of functional programming for real-world applications. Obviously, there is not much use in us academics telling them otherwise. Instead, we had the idea to invite people from industry to offer functional programming workshops about practical topics not covered in our course.

In WS20, we were able to host three workshops on 1) design patterns for functional programs 2) networking and advanced I/O and 3) user interfaces and functional reactive programming. We limited participation to 35 students for each workshop, and to our delight, demand exceeded supply (more than 120 students applied). Both industry partners and workshop participants reported very positively to us. In some cases, workshops were even extended for multiple hours due to the great curiosity by students. Moreover, organisational overhead was small: we merely had to communicate the syllabus to our partners and coordinate time and place. We envisage offering even more workshops in future iterations and highly recommend this mechanism to demonstrate real-world applicability to other educators.

Social Interactions Studies confirmed that the COVID-19 pandemic worsened students' social life, leading to higher levels of stress, anxiety, loneliness, and symptoms of depression [6]. This got us thinking about mechanisms we can employ in WS20 to foster social interaction and exchange between students – which also play a crucial pedagogical role in general [7].

Firstly, we decided to employ pair-programming (groups of 3–4 students) in our online tutorials. The technical setup for this is described in Section 4.2. This not only made social interactions an integral part of the tutorial, but also had positive effects on knowledge sharing. We can report very positively on this policy.

Secondly, we hosted two informal get-together sessions, one at the beginning and one at the end of

the semester. We started with icebreaker sessions in breakout rooms, randomly allocating students and at least one student assistant in each group. Following these sessions, we opened thematic breakout rooms where students could freely move and talk about a given topic. Some students preferred to talk about the course material, others had light-hearted conversations about university, yet others started to play online games. All in all, we received very positive feedback for our get-together sessions.

Thirdly, we organised an ACM-ICPC-like programming contest (see Section 4.3) where students could participate in teams, followed again by a light-hearted get-together session for participants.

4.2 Technical Setup and Automated Assessment

Automated Assessment In WS19, we used an improved version of the testing infrastructure introduced in [2]. However, this system was not able to manage online exams nor to mark non-programming tasks. We thus switched to a newly written open-source tool developed at TUM called ArTEMiS [10]. ArTEMis is a highly scalable, automated assessment management system and is programming language independent – it only expects test runners to produce tests results adhering to the Apache Ant JUnit XML schema. It already offered support for a few imperative programming languages, and we added support for Haskell[10].

As ArTEMis takes care of most things, including automated test execution and score management, and offers an exam mode and good support for grading non-programming tasks, the only thing we were left to do was writing the actual test code. For the most part, we verified the results computed by a student's submission by comparing them to those computed by a sample solution written by us. In some cases, we also tested for efficiency using timeouts. Our tests were powered by the following libraries:

1. QuickCheck [3]: Although QuickCheck provides means to automatically generate test data (using the typeclass `Arbitrary`), most tests and benchmarks used custom QuickCheck input generators. This was necessary to increase coverage and eliminate non-applicable inputs for tests with preconditions. We also used custom shrinkers to provide better feedback to students in case of a failure. In both cases, the flexible combinators provided by QuickCheck made this a straightforward task.
2. SmallCheck [15]: The exhaustive testing facilities provided by SmallCheck mainly served as a complementary tool that provided small counterexamples for, in many cases, obvious deficiencies.
3. Tasty[11]: We used Tasty to put QuickCheck, SmallCheck, and unit tests as well as the checking of "Check Your Proof" proofs (see Section 5) into one common framework that is capable of generating results interpretable by ArTEMiS. We used the unit testing facilities of Tasty to complement our QuickCheck/SmallCheck tests with corner cases. Integration of proof checking was pleasantly straightforward. Moreover, Tasty supports timeouts for individual test cases, solving the issue of truncated test reports mentioned in [2].

Development Environment and Online Tutorials In previous iterations, there were no recommendations for students regarding the development environment they should use for the practical part of the course. However, due to the COVID-19 pandemic, this was no longer an option: we needed a way for students to share their code in read and write mode to other peers (pair-programming) and their teaching assistant (for feedback purposes) during online tutorials. Moreover, as explained in Section 4.1, we decided to give up on manual feedback on code quality and wanted students to use a linter instead. Finally, we had negative experiences with students installing no compiler at all and instead (mis)using our

[10] It now also supports OCaml
[11] https://hackage.haskell.org/package/tasty

submission server as a compiler backend. We thus introduced a strict policy for the technical setup to be used during the tutorials. Detailed installation instructions can be found online[12]. Here we briefly list the key components and our experiences:

1. IDE: We used VSCodium[13] due to its cross-platform support, rich library of extensions, widespread adoption, and free open-source software philosophy. We did not receive any negative feedback by students and, besides few exceptions, no major installation problems were reported.
2. Build and dependency manager: We used Stack[14] for these purposes. Since Stack provides curated sets of packages and compiler versions that are checked for compatibility, deterministic builds are guaranteed. Students hence showed little struggle to compile and execute their programs.
3. Linter: We used HLint[15], which, among other things, provides suggestions for alternative functions and simplified code and spots redundancies. Students reacted curiously and positively to these suggestions. In particular, simplified code fragments were discussed vividly among pair-programming teams.
4. API search engine: Due to Haskell's strong type system, searching its API by type signature often returns better results than searching for function names. For this purpose, we introduced Hoogle[16] to our students and let them install an extension that integrates such searches into their IDE. Sadly, we have no data to report on whether this enriched students programming experiences.
5. Real-time collaboration: Finally, we used VSLiveShare[17] to host our pair-programming sessions (in groups of 3–4 students) during the online semester. We can report positively on its connection stability and usability. Unfortunately, the plugin requires a Microsoft or GitHub account, but since almost all students already signed up to the later before, this requirement passed uncontroversially. Nevertheless, for future iterations, we plan to investigate in open-source alternatives.

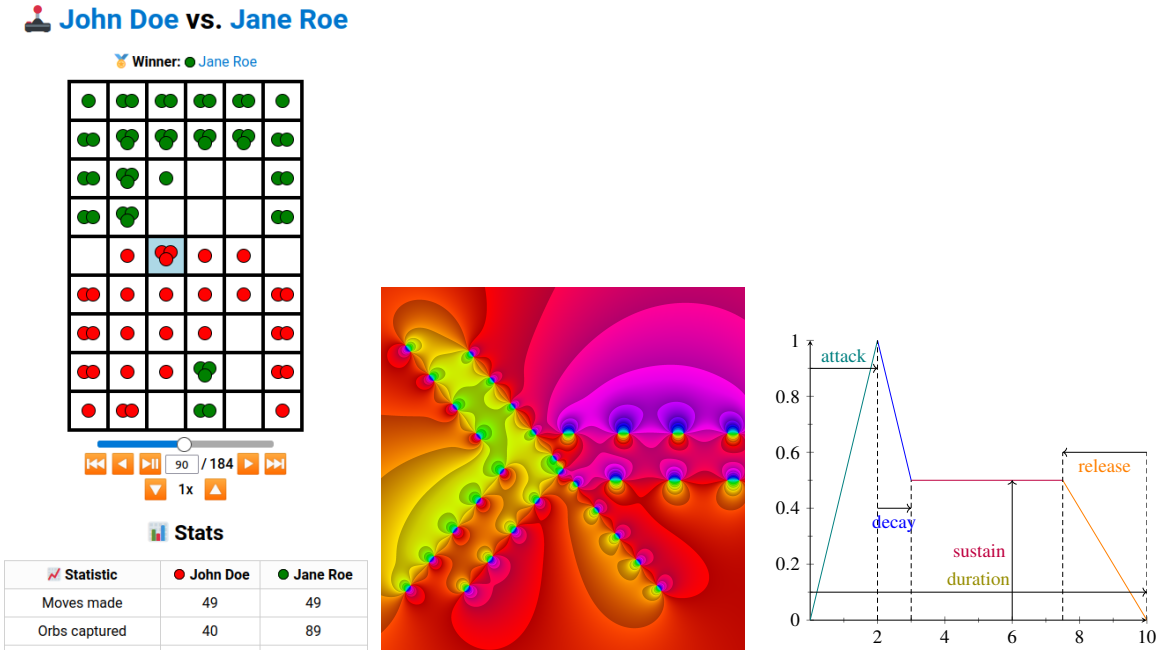
All in all, we can report positively on the setup. We have experience with running online tutorials for other large lectures and can testify that the system in this paper worked best in technical as well as social aspects. We can also report that students were more knowledgeable about their Haskell programming environment (compilation, dependency management, etc.) than in previous iterations. Misappropriation of the submission server as a compiler backend also stopped.

4.3 Selected Exercises and Tools

Many students at TUM have questioned the applicability and usefulness of functional languages after completing their mandatory functional programming course. We believe this is mainly due to two reasons: 1) introductory programming courses often stick to simple algorithmic or mathematically inspired challenges and 2) side-effects (in particular I/O) are often introduced very late in functional programming courses.

Changing the latter appeared unpromising to us: we think that students would be confused if a "special" `IO` type and do notation were to be introduced before they are comfortable with the basic features of functional languages. We thus cranked the other handle by creating diverse exercises that go beyond

[12] https://www21.in.tum.de/teaching/fpv/WS20/installation.html
[13] VSCodium provides free open-source software binaries of VSCode https://vscodium.com/
[14] https://www.haskellstack.org/
[15] https://github.com/ndmitchell/hlint
[16] https://hoogle.haskell.org/
[17] https://visualstudio.microsoft.com/services/live-share/



(a) Game strategies and tournaments

(b) Art generation

(c) Music synthesis

Figure 1: Examples of exercises created as part of the course

simple terminal applications. Designing and implementing such exercises, however, is labour-intensive. As mentioned in Section 4.1, we thus decided to reallocate resources and let our student assistants help us with this work rather than providing feedback for homework submissions.

This turned out to be a very fruitful idea: the quality of our student assistants' work was often way above what we expected. The one difficulty we initially faced was the mediocre quality of tests written by most assistants. They only had the rudimentary knowledge of QuickCheck taught as part of the course. We thus hosted a workshop for our assistants that explained our testing infrastructure and provided best-practice patterns when writing tests. The quality of tests significantly increased following this workshop, though we still had to polish them before publication.

We next introduce a few exercises and tools that were created as part of the course. They are available in this article's repository[18], next to our other exercises, including a music synthesiser framework, a turtle graphics framework, an UNO framework, and guided exercises for DPLL and resolution provers. Some further examples can be found on our competition blogs[19].

Game Tournament Framework It has become an annual tradition in the course to run a game tournament over the Christmas break. In this tournament, the students are tasked with writing an AI for a board game that competes against the AIs of their fellow students. To pass the homework sheet, it suffices to implement a basic strategy, but to score well in the competition, students have come up with quite sophisticated strategies in the past years. The framework allows the strategies to use statefulness and

[18] https://github.com/kappelmann/engaging-large-scale-functional-programming

[19] https://www21.in.tum.de/teaching/fpv/WS20/wettbewerb.html (WS20) and https://www21.in.tum.de/teaching/fpv/WS19/wettbewerb.html (WS19)

randomization, so that there are few limits to the students' creativity. The best strategies usually use the Minimax rule with or without alpha-beta-pruning and a clever evaluation function, which is particularly important in this setting because the game tree cannot be evaluated very far given the limited resources for each submission in the tournament.

The tournament runs continuously for 2–3 weeks and the results are displayed on a website (see Figure 1a for an example from the 2020 iteration). Students thus get reasonably quick feedback on how their strategy performs, which keeps them engaged and allows them to improve their strategy iteratively. The tournament has become a popular feature of the course with 182 participating students in WS19 and 220 in WS20.

In our repository, we provide the framework along with code specific to the game from WS20, which is based on Chain Reaction [20]. It runs a round-robin tournament, collecting statistics for each game and player. Instructions for adapting the framework to a different game can also be found in the repository.

Programming Contest Framework To foster social interaction and diversify our bonus system, we hosted an ACM-ICPC-like programming contest. In such contests, students participate in teams of 2–3, solving as many programming challenges as possible in a given time frame, and can check their ranking on a live scoreboard. At some point during the contest, the scoreboard gets frozen, and following the working time, solutions to all challenges are presented by the organisers. The final results are then revealed by unfreezing the scoreboard again.

We found existing solutions to run such contests too heavyweight for our purpose and hence created a lightweight alternative. Our framework continuously receives test results, computes each team's score, and displays the live scoreboard and task instructions. It is agnostic to the programming language and test runner used. It expects tests results adhering to the Apache Ant JUnit XML schema, but modifying it to support other formats would be straightforward. Detailed deployment instructions can be found in this article's repository.

We ran an online iteration of the contest in WS20, again using ArTEMiS as a test runner. Teams were cooperating on their platform of choice and were able to ask for clarifications on a dedicated online channel. Our experiences are very positive: a total of 27 teams participated in the contest and most stayed for the social hangout following it. Given the presented framework, the technical setup of the contest requires little time. Some significant time, however, must be spent on setting up the challenges, tests, and solutions, though plenty of challenges may be found online by searching for other contests, which one then may modify and reuse. In general, we recommend running such contests for every lecture concerned with programming concepts.

I/O-Mocking Library As discussed in Section 4.2, we primarily use QuickCheck to automatically assess submissions for homework exercises. This raises the question how monadic I/O in Haskell can be tested on the submission system. Since we do not want to actually execute the side effects that the submitted code produces, the obvious solution is to use a mocked version of Haskell's `IO` type.

A standard approach to mock `IO`, which is put forward by packages such as `monad-mock`[21] and `HMock`[22], is to first extract the side effects that are required for a certain computation into a new typeclass. Since a typeclass allows multiple instantiations, we can then provide one instantiation that actually executes the side effects on the machine and another one that just modifies a mocked version of the

[20] https://brilliant.org/wiki/chain-reaction-game/
[21] https://hackage.haskell.org/package/monad-mock
[22] https://hackage.haskell.org/package/HMock

environment. For example, to implement a function that copies a file, we need two operations: one for reading a file and one for writing a file.

```
import qualified Prelude
import Prelude hiding (readFile, writeFile)

class Monad m => MonadFileSystem m where
  readFile :: FilePath -> m String
  writeFile :: FilePath -> String -> m ()
```

The implementation is straightforward.

```
copyFile :: MonadFileSystem m => FilePath -> FilePath -> m ()
copyFile source target = do
  content <- readFile source
  writeFile target content
```

Due to the definition of `MonadFileSystem`, the instance for `IO` is trivial. The mocked version can be implemented as a map from file names to file contents wrapped by the `State` monad transformer to make it mutable. We omit the instantiation of `MonadFileSystem` for brevity. Testing `copyFile` is now as simple as checking whether the state of the file system is as expected after executing the function. An example that includes the instance `MonadFileSystem MockFileSystem` and a test can be found in the repository in resources/io_mocking/typeclass.

```
instance MonadFileSystem IO where
  readFile = Prelude.readFile
  writeFile = Prelude.writeFile

data MockFileSystem = MockFileSystem (Map FilePath String)
instance MonadFileSystem (State MockFileSystem) where
  readFile = ...
  writeFile = ...
```

While this approach is sufficient for many use cases, it lacks one important property: transparency. More specifically, the code submitted by students must contain or import `MonadFileSystem` and the signatures of terms that use `IO` must be adapted. This is especially problematic because the lecture introduces `IO` without mentioning monads, which are only introduced towards the end of the course.

Instead of modifying existing code, we delay the mocking to a later stage, namely the stage of compilation. We achieve this with a mixin that replaces the `IO` module of a submission with a mocked version. The mocked `IO` type can be realised in a similar way as above mock file system. However, to achieve full transparency this time, we not only need a file system but also handles, such as standard input and output, as well as a working directory.

All these aspects of the machine state are summarised in below type `RealWorld`. Crucially, the type also contains a mock user, represented by a computation of type `IO ()`, which interacts with a student's submission; that is, the user generates the input for and reads the output of the student's submissions. For simplicity, we do not show the full type here. As before, this type is wrapped by the `State` monad transformer as well as two additional transformers `PauseT` and `ExceptT` in order to form the mocked `IO` type.

```
data RealWorld = RealWorld {
  workDir :: FilePath,
  files :: Map File Text,
  handles :: Map Handle HandleData,
  user :: IO (),
  ...
}

newtype IO a =
  IO { unwrapIO :: ExceptT IOException (PauseT (State RealWorld)) a }
```

While the transformer `ExceptT` simply adds I/O exceptions, such as errors for insufficient permissions, the purpose of `PauseT` is not obvious. To understand its role, consider the following simple program that reads the user's name and greets them.

```
module Hello where

main = do
 name <- getLine
 putStrLn $ "Hello " ++ name
```

In a normal (non-mocked) execution of the program, the program blocks and waits for input when `getLine` is called. If our mocked `IO` type would only consist of a state monad, all the input to the program would have to be passed in one monolithic unit. However, programs may consume input multiple times. We thus need a way to suspend the program every time a blocking operation is called and transfer control over to our mock user. The mock user then reacts to the output of the program and generates the input that the program is waiting for. When the user is done, it yields and the program of the student is resumed.

These considerations lead us to the monad below, consisting of two operations: the first one pauses execution whereas the second one runs a computation of the monad until either `pause` is called or the computation finishes. In the former case, `stepPauseT` returns a `Left c` where `c` represents the rest of the computation; in other words, the part of the computation that is executed when resuming. Otherwise, the final result `r` of the computation is returned as `Right r`. It should be noted that the pause monad is an instance of the more general coroutine monad as provided by the `monad-coroutine`[23] package. For the implementation details of the corresponding monad transformer `PauseT` we refer to the repository.

```
class Monad m => MonadPause m where
  pause :: m ()
  stepPauseT :: m a -> m (Either (m a) a)
```

We exemplify the mechanics of the mocking framework with a simple test of the `main` function from above. To this end, we first implement a mock user that takes a name and supplies it to the standard input of `main`. The user then reads the output of the program and checks whether it printed the expected greeting. In the QuickCheck property `prop_hello`, we evaluate the interaction between the mock user and the program with `Mock.evalIO` on `Mock.emptyWorld`, a minimal `RealWorld` that contains no files and only the absolutely necessary handles: standard input, standard output, and standard error. The interaction itself sets the user to `user s`, then executes the `main` function, and finally runs the user to completion.

[23] https://hackage.haskell.org/package/monad-coroutine

```
module Test where

import qualified Mock.System.IO.Internal as Mock
import qualified Hello as Sub

user :: String -> Mock.IO ()
user s = do
  Mock.hPutStrLn Mock.stdin s
  output <- Mock.hGetLine Mock.stdout
  when (output /= ("Hello " ++ s))
    (fail $ "\nExpected:\n" ++ "Hello " ++ s
      ++ "\nActual:\n" ++ output ++ "\n")

prop_hello = forAll (elements ["Karl", "Friedrich", "Rosa"]) $ \s ->
  Mock.evalIO (Mock.setUser (user s) >> Sub.main >> Mock.runUser)
              Mock.emptyWorld
```

Figure 2 illustrates the evaluation steps of `Mock.evalIO`. Note that there are two blocking operations (i.e. operations that call `pause` internally), namely `getLine` and `Mock.hGetLine Mock.stdout`. When `Mock.evalIO` encounters any such operation, it transfers control between the user and the program as indicated by the black arrows. Control is also transferred if the computation runs until completion without meeting a `pause`. The horizontal axis with white arrows illustrates the return values of `stepPauseT`. Focussing on the `main` function, we see that `stepPauseT` returns the remaining computation `putStrLn $ "Hello " ++ x` wrapped in a `Left` when encountering `getLine`. After the user provides the input for `getLine` and yields, the `main` function prints the greeting and finishes with the result `Right ()`. Similarly, the user is blocked on `Mock.hGetLine`, which means that the remaining computation only consists of the `when` check, which is executed as soon as `main` is done. This explains why we need to run `Mock.runUser` after `Sub.main` since the crucial `when` check would never be executed otherwise.

All in all, the mocking framework lets us uniformly test student submissions with common frameworks like QuickCheck and Smallcheck regardless of whether they contain I/O effects.

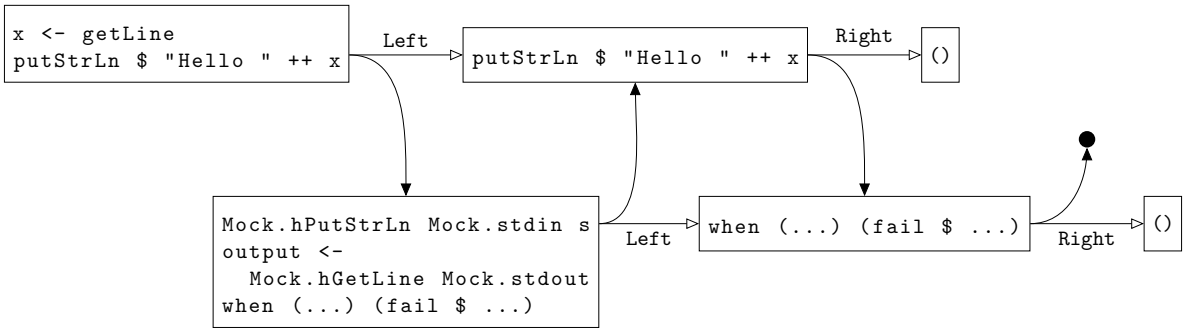


Figure 2: Interaction between the mock user and the student's submission. White arrows indicate the return value of `stepPauseT` whereas black arrows indicate transfer of control. The black dot signifies the end of the interaction.

5 Check Your Proof by Example

Besides functional programming, the course also dealt with the verification of functional programs. Even though we only consider a strict and total subset of Haskell for our proofs, equational reasoning together with induction (and case analysis) is already sufficient to prove interesting properties. Since "fast and loose reasoning is morally correct" [4], valid properties in this sub-language carry over to Haskell.

Simple inductive proofs of course lend themselves well to be automatically checked and, as announced in [2], a tool called "Check Your Proof" (CYP for short) was developed at our lab by Durner and Noschinski[24].

The first example of such a proof presented in the lecture is the proof of associativity of the append-operator for lists. Proving this example in CYP first requires us to define the data type of lists. We use the constructors `[]` and `:` as is standard in Haskell.

```
data List a = [] | a : List a
```

Now we can define the infix append-operator `++`

```
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
goal xs ++ (ys ++ zs) .=. (xs ++ ys) ++ zs
```

and state the goal

```
xs ++ (ys ++ zs) .=. (xs ++ ys) ++ zs
```

where all variables are implicitly universally quantified. The listings above describe the background theory of the proof. The theory is fixed in a file and provided to the students by the instructors. The students then supply the proof in a separate file. In our case, we proceed to prove the statement by structural induction on `xs`.

```
Lemma: xs ++ (ys ++ zs) .=. (xs ++ ys) ++ zs
Proof by induction on List xs
  Case []
    To show: [] ++ (ys ++ zs) .=. ([] ++ ys) ++ zs
    Proof
                      [] ++ (ys ++ zs)
      (by def ++) .=. ys ++ zs
      (by def ++) .=. ([] ++ ys) ++ zs
    QED
```

In above listing, the first `Proof` marks the beginning of an inductive proof whereas the second `Proof`, which has no further arguments, starts an equational proof. While CYP allows to arbitrarily nest proofs by case analysis or induction, the innermost goal must always be discharged by an equational proof. An equational proof is a chain of equations that rewrite the left-hand side of the current goal to the right-hand side. The user has to justify each rewrite step by a corresponding equation that yields the right-hand side when applied to the term or a subterm on the left-hand side. Note that in the example `(by def ++)` refers to either one of the defining equations of `++`, and CYP will check if any of them justifies the current step.

Now, consider the inductive case.

[24] https://github.com/noschinl/cyp

```
  Case x : xs
    To show: (x : xs) ++ (ys ++ zs) .=. ((x : xs) ++ ys) ++ zs
    IH: xs ++ (ys ++ zs) .=. (xs ++ ys) ++ zs
    Proof
                          (x : xs) ++ (ys ++ zs)
      (by def ++)    .=. x : (xs ++ (ys ++ zs))
      (by IH)        .=. x : ((xs ++ ys) ++ zs)

                          ((x : xs) ++ ys) ++ zs
      (by def ++)    .=. (x : (xs ++ ys)) ++ zs
      (by def ++)    .=. x : ((xs ++ ys) ++ zs)
    QED
QED
```

Again, CYP requires the user to be explicit: the goal in each inductive case and the induction hypotheses have to be spelled out. Note that CYP also allows one to start rewriting from the left-hand side as well as the right-hand side of the goal.

The lecture then goes beyond proofs by structural induction and introduces proof by extensionality, case analysis, and computation induction, all of which CYP supports with some conditions applying as we will see shortly. Computation induction in particular was not supported in the original version by Durner and Noschinski but was only introduced in a fork by us in WS20[25]. CYP also allows proving named auxiliary lemmas, which is useful to modularise proofs and often necessary to prove generalised versions of a given goal.

The simplicity of CYP both in its usage and its implementation comes with some caveats:

- The version of CYP used by us is untyped and thus unsound if the background theory contains multiple types as demonstrated in [14]: given a singleton type `data U = U` in the background theory, one can prove `x .=. y` by case analysis. In an untyped environment, one can then use this lemma to prove equality between any two terms.
- Barring soundness issues due to a lack of types, one also needs to ensure that all function definitions are total and that their patterns do not overlap.
- Computation induction is unsound if there are recursive calls in branches of an if-then-else expression. This is because the induction hypotheses would have to be conditional rewrite rules in such cases, which CYP does not support.

Renz et al. [14] discuss the inner workings of CYP in detail and develop an extension to CYP that introduces types, thus solving the first issue. They also made it possible to leave holes in proofs and in expressions, which then have to be filled in by students.

Solving the other issues, however, would incur additional effort. We think that CYP should be put on a stronger foundation such as higher-order logic (HOL) without compromising its simplicity from a user perspective. One possibility would be to rewrite CYP as as a frontend to Isabelle/HOL [13]. Since HOL is a typed logic, it would solve the first issue. The latter issues could be resolved using Isabelle's function package [9]. A stronger foundation also allows one to more confidently develop new extensions for CYP.

Our teaching experience with CYP has been very positive. CYP proof checking is quick and can easily be integrated in the testing framework (Tasty) that we used for our programming exercises. This allowed us

[25] https://github.com/lukasstevens/cyp

to deal with programming and proof exercises in a uniform and scalable way. CYP was also generally liked by our students as their feedback testifies: in WS19, we received two positive comments and one negative comment without asking for feedback on CYP. When asked explicitly about their thoughts on CYP in WS20, the students answered with 18 positive comments and three negative comments. The students liked the instantaneous feedback that CYP provides, which helped them to deepen their understanding of inductive proofs at their own pace. The main criticism of CYP was the lack of documentation. A CYP-cheatsheet developed by a tutor was later added to the repository of CYP to improve the situation.

In each of the exams, two exercises (out of ≈ 8) were concerned with inductive proofs. The first exercise was a straightforward proof by structural induction while the second one required more effort, e.g. generalisation of the goal or a slightly less trivial computation induction. We did not require the students to stick to CYP's syntax in the exam, but we urged them to follow a similar structure, which worked very well overall and simplified the grading.

6 Exams

In previous iterations of the course, exams had been paper-based by necessity as university regulations made digital exams unfeasible. Due to the COVID-19 pandemic, however, remote exams (either unsupervised or supervised via video conferencing software) were allowed for the repeat exam in WS19 and both exams in WS20.

In the following, we will outline how we adapted our exam process to this new reality and the advantages of our approach for students and staff.

Workflow for Students We chose the ArTEMiS [10] platform, previously described in Section 4.2, for the exams. In WS19, students were mostly unfamiliar with ArTEMiS, but in WS20, they had been submitting their homework using the same workflow as during the exam.

Students were able to check out individual exam questions from the ArTEMiS repository and then work on them using their preferred programming environment. In addition to programming exercises, the exams included theory questions and proofs, which were submitted as text using an online editor.

The exams were open-book, including online resources but prohibiting posting questions to chats, forums, etc. Usage of third-party code had to be cited using comments. We feel that the combination of a programming environment customised to the individual student's taste and access to online resources comes close to how programming is done in practice.

In contrast, previous exams were purely paper-based. This limited the scope of programming exercises we were able to pose since both writing and grading programs on paper is highly labour-intensive. Moreover, we were able to expect a higher standard of correctness since minor errors, in particular in syntax, are excusable when programming on paper but less so when students can test, or at least run, their programs before submission.

Grading Grading hundreds of exams on paper is a huge undertaking. In previous iterations, it took 10–20 people (staff and student assistants) about 4–5 days to grade an exam. Additionally, grading programming exercises on paper is error-prone, and it is unfeasible to digitise every paper submission and run it through a compiler.

Using ArTEMiS, we were able to use automated testing for programming exercises. In most cases, we were able to mark submissions which passed all tests as fully correct and manual grading was only needed for submissions that failed some tests or did not compile. In our view, manual grading is still necessary in

those cases because it is generally not possible to write tests that are fine-grained to such a degree that every potential source of error is recognised and scored proportionally. This hybrid approach to grading still significantly lightened the workload, for even manual grading is much more convenient when there are test results for guidance and one can re-run tests after fixing compilation or minor semantic errors.

Online Review Once graded, students have the right to review their exams in order to check for errors or unfairness in the grading process. For paper-based exams, students usually had to make an appointment for a time-slot to review their exam under the supervision of teaching staff. In courses with a high number of participants, this imposed a significant organisational overhead for both staff and students.

Using ArTEMiS, however, every student can simply review their exam and submit complaints through an online interface. After the review period is over, the complaints were approved or rejected by the teaching staff. This resulted in a vastly increased proportion of students reviewing their exam, thus ensuring a higher degree of fairness in the grading process.

In addition, feedback for students is improved by this method since they are able to check test results along with in-line grading comments in the exam review instead of having to rely on handwritten annotations made by the corrector.

Cheating University regulations allow both supervised or unsupervised remote exams. We considered supervision but ultimately decided against it. Were we to supervise the exam ourselves, each staff member would have to keep track of 20–30 students via their webcam in a video conferencing software. We feel that this would hardly ensure against cheating since, for example, students would still be able to take advice from someone out of view of the camera or communicate with others online (it does not seem feasible to simultaneously supervise the camera and screen-share of 20–30 students).

There are also commercial options for exam supervision, but it is unclear how effective they are at preventing cheating, and the options we reviewed raised significant privacy concerns.

We thus decided to rely on an honour system for our exams. Beyond the honour pledge, we also checked for plagiarism with Moss. This turned up only three cases of conclusive plagiarism. We suspect and accept that there were likely more cases of cheating that we could not catch, so we aim to improve our anti-cheating measures in future iterations.

We also created 3–4 slight variations of each exam question in order to make plagiarism slightly more onerous. ArTEMiS supports the creation of these variants and assigns them randomly during the exam.

7 Conclusion

As computer science departments continue to grow, COVID-19 continues to spread, and imperative programming appears to be the industry norm, we functional programming educators have to find ways to make our courses more scalable and engaging, while demonstrating the elegance and usefulness of functional programming and having to adapt to a mix of teaching in physical and virtual space. We hope our efforts not only inspired our 2000 students but also other educators to take on these challenges. The insights and resources presented in this article proved valuable to us and will hopefully also do to others.

For future iterations, we plan to keep many things that we introduced during the pandemic and double down on some new engagement mechanisms such as the supplementary workshops. One thing we want to improve is the time we have to spend on the weekly competition, for example by handing some of the workload to student assistants and creating fewer but more engaging competition exercises. Another possibility we want to explore is offering more competition exercises that can be automatically

graded and results continuously be published on a ranking website (cf. our Chain Reaction competition in Section 4.3). Besides reducing the time we have to spend on evaluating submissions, this would also increase engagement due to instant feedback. We also plan to stick to the digital open-book exam format, though as of yet, we have no good solution to prevent cheating.

Acknowledgements We want to thank all people involved in the course, in particular Tobias Nipkow, Manuel Eberl, our student assistants, the ArTEMiS development team, our industry partners Active Group, QAware, TNG Technology Consulting, and Well-Typed, and last but not least our 2000 Haskell students.

References

[1] *Facts and Figures – TUM Department of Informatics*. https://web.archive.org/web/20211010121452/https://www.in.tum.de/en/the-department/profile-of-the-department/facts-figures/. Accessed: 2021-12-22.

[2] Jasmin Christian Blanchette, Lars Hupel, Tobias Nipkow, Lars Noschinski & Dmitriy Traytel (2014): *Experience Report: The next 1100 Haskell Programmers*. *SIGPLAN Not.* 49(12), p. 25–30, doi:10.1145/2775050.2633359. Available at https://doi.org/10.1145/2775050.2633359.

[3] Koen Claessen & John Hughes (2011): *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. *SIGPLAN Not.* 46(4), p. 53–64, doi:10.1145/1988042.1988046. Available at https://doi.org/10.1145/1988042.1988046.

[4] Nils Anders Danielsson, John Hughes, Patrik Jansson & Jeremy Gibbons (2006): *Fast and Loose Reasoning is Morally Correct*. *SIGPLAN Not.* 41(1), p. 206–217, doi:10.1145/1111320.1111056. Available at https://doi.org/10.1145/1111320.1111056.

[5] John Dunlosky, Katherine A. Rawson, Elizabeth J. Marsh, Mitchell J. Nathan & Daniel T. Willingham (2013): *Improving Students' Learning With Effective Learning Techniques: Promising Directions From Cognitive and Educational Psychology*. *Psychological Science in the Public Interest* 14(1), pp. 4–58, doi:10.1177/1529100612453266. arXiv:https://doi.org/10.1177/1529100612453266. PMID: 26173288.

[6] Timon Elmer, Kieran Mepham & Christoph Stadtfeld (2020-07): *Students under lockdown: Comparisons of students' social networks and mental health before and during the COVID-19 crisis in Switzerland*. *PLoS ONE* 15(7), p. e0236337, doi:10.3929/ethz-b-000428501.

[7] Beth Hurst, Randall R. Wallace & Sarah B. Nixon (2013): *The Impact of Social Interaction on Student Learning*. *Reading Horizons* 52, pp. 375–398.

[8] Nate Kornell (2009): *Optimising learning using flashcards: Spacing is more effective than cramming*. *Applied Cognitive Psychology* 23(9), pp. 1297–1317, doi:https://doi.org/10.1002/acp.1537. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/acp.1537.

[9] Alexander Krauss (2009): *Automating recursive definitions and termination proofs in higher-order logic*. Ph.D. thesis, Technical University Munich. Available at http://mediatum2.ub.tum.de/doc/681651/document.pdf.

[10] Stephan Krusche & Andreas Seitz (2018): *ArTEMiS: An Automatic Assessment Management System for Interactive Learning*. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, Association for Computing Machinery, New York, NY, USA, p. 284–289, doi:10.1145/3159450.3159602. Available at https://doi.org/10.1145/3159450.3159602.

[11] Prashant Loyalka, Ou Lydia Liu, Guirong Li, Igor Chirikov, Elena Kardanova, Lin Gu, Guangming Ling, Ningning Yu, Fei Guo, Liping Ma, Shangfeng Hu, Angela Sun Johnson, Ashutosh Bhuradia, Saurabh Khanna, Isak Froumin, Jinghuan Shi, Pradeep Kumar Choudhury, Tara Beteille, Francisco Marmolejo & Namrata Tognatta (2019): *Computer science skills across China, India, Russia, and the United States*.

Proceedings of the National Academy of Sciences 116(14), pp. 6732–6736, doi:10.1073/pnas.1814646116. arXiv:https://www.pnas.org/content/116/14/6732.full.pdf.

[12] National Academies of Sciences, Engineering, and Medicine (2018): *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments*. The National Academies Press, Washington, DC, doi:10.17226/24926. Available at https://www.nap.edu/catalog/24926/assessing-and-responding-to-the-growth-of-computer-science-undergraduate-enrollments.

[13] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9. Available at https://doi.org/10.1007/3-540-45949-9.

[14] Dennis Renz, Sibylle Schwarz & Johannes Waldmann (2020): *Check Your (Students') Proofs-With Holes*. *CoRR* abs/2009.01326. arXiv:2009.01326.

[15] Colin Runciman, Matthew Naylor & Fredrik Lindblad (2008): *Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values*. In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, Association for Computing Machinery, New York, NY, USA, p. 37–48, doi:10.1145/1411286.1411292. Available at https://doi.org/10.1145/1411286.1411292.