Extended Abstract: How to Derive an Electronic Functional Programming Exam from a Paper Exam with Proofs and Programming Tasks

Ole Lübke[*] Konrad Fuger[†]
Fin Hendrik Bahnsen[‡,§] Katrin Billerbeck[¶] Sibylle Schupp[*]
{ole.luebke, k.fuger, fin.bahnsen, katrin.billerbeck, schupp}@tuhh.de
[*]Institute for Software Systems
[†]Institute of Communication Networks
[¶]Center for Teaching and Learning
Hamburg University of Technology (TUHH)
Hamburg, Germany
[‡]Institute for Artificial Intelligence in Medicine
University Medicine Essen
Essen, Germany

Electronic exams (e-exams) have the potential to substantially reduce the effort required for conducting an exam through automation. Yet, care must be taken to sacrifice neither task complexity nor constructive alignment in favor of automation. We show how an e-exam for functional programming can be derived from a paper exam with proof and programming tasks in a way that accounts for the above-mentioned trade-off. As a first step, we analyzed our pre-e-exam course and exams to identify potentials for automation, keeping in mind the learning objectives of the course and how tasks are aligned with them. We then devised a plan for the transformation that we realized by extending the YAPS e-exam system. The extensions include a new standalone tool that analyzes student code for task-relevant features, checking answers with regular expressions, a new algorithm for evaluating proofs, and a general-purpose comment field for students. Additionally, we created a new higher-level language to specify regular expressions tailored to common patterns in Haskell code. We evaluated the resulting e-exam by analyzing the degree of automation in the grading process, asking students for their opinion, and critically reviewing our own experiences. We found that almost all tasks can be graded automatically at least in part, and correct solutions can often be detected as such. Yet, awarding partial points is still often impossible. The students agree that an e-exam is the right examination format for the course, and examiners (us) enjoy a more time-efficient grading process. On the other hand, creating an e-exam requires more effort than creating a paper exam, so in the future we also want to investigate ways to introduce more automation in that phase.

Keywords: Electronic Examination, Automated Grading, Constructive Alignment

1 Introduction

A strong argument for electronic exams (e-exams) is their high potential for automation, which can decrease the effort of conducting an exam, especially during grading. Yet, care must be taken to sacrifice neither task complexity [6] nor constructive alignment (CA) [3] in favor of automation. However, e-exams also offer a great opportunity to improve the alignment between learning objectives, activities, and assessment. It is essential that digital teaching should likewise be reflected in digital exams rather than testing programming skills on paper. Still, there is a challenge in developing appropriate electronic, complex, and open-ended tasks to assess deeper understanding. Following the recent introduction of large scale electronic examinations [7] and the development of the extensible *Your Open*

[§]F. H. Bahnsen was with the Institute of Embedded Systems, TUHH, when the presented work was created.

Submitted to:
TFPIE 2023

Examination System for Activating and emPowering Students (YAPS) [1] at Hamburg University of Technology (TUHH), we show how to develop and implement high-quality, professionally appropriate, and cognitively demanding e-exam tasks in the functional programming (FP) course at TUHH. YAPS enables competency-based testing and is thus fully in line with the principles of CA. It is licensed open source and follows a contactless and state-based operating concept that reproduces the testing procedure of TUHH. Because of data protection laws in Germany there is no real alternative to self-hosting the examination software, which excludes many alternative examination systems.

This extended abstract is organized as follows: Section 2 summarizes how we analyzed our pre-e-exam FP courses and past exams to identify potentials for automation and improvement. Section 3 showcases how we realized our ideas, mostly through extending YAPS. Because we teach FP using the example of Haskell but YAPS only offered a C/C++ compiler, we extended it with the Glasgow Haskell Compiler (GHC)[1]. Additionally, the extension features a tool to analyze student code for task-relevant features (e.g., usage of pattern matching) that we developed specifically for the e-exam. Another extension is checking answers with regular expressions (RegExs). We found that crafting RegExs that are flexible enough to accept all valid solutions, but also strict enough to reject any wrong answers, is a challenging and time-consuming process. Therefore, we devised a small, high-level specification language for RegExs tailored to common patterns found in Haskell code, and a tool that compiles these specifications to actual RegExs. Furthermore, YAPS features a way for students to enter proofs ("Proof Puzzle") that are then graded fully automatically. However, the original algorithm produces results that differ from our own, manual evaluation. Consequently, we developed a new algorithm based on the length of correct proof sequences that is congruent with our judgment. Finally, while a paper exam is very flexible in terms of input (i.e., anything that can be written or drawn), an e-exam is not because it requires specific input in specific input fields (e.g., a numeric input field only accepts numbers). Because we did not want to take that flexibility away from students, we extended YAPS with a comment field for each task that students can freely use to their liking. Section 4 contains the evaluation with respect to degree of automation, the opinions of students collected in a poll directly after the exam, and the opinions of the examiners (us). Section 5 provides a summary and an outlook on future work we plan for the e-exam.

2 Analysis of our Pre-E-Exam FP Course

Changes to the examination of a course should be reflected in the teaching activities, and vice versa. This ensures that the teaching activities prepare the students to take the exam, and that the exam assesses the knowledge and skills taught in the course. To be able to review this mutual dependency, in this section we summarize the analysis of our pre-e-exam FP course and exams.

Overall, there are three weekly teaching activities in the course: First, the lecture lays the foundation for the other activities. Second, so-called programming labs provide an opportunity to get hands-on experience with FP. During the labs, students solve small programming tasks with the support of student tutors [5, 2]. Third, there are homework exercise sheets that, in contrast to the labs, are supposed to be solved alone. The tasks in labs and exercises match the topics covered in the lecture during the same week and are designed to support the students in reaching the learning objectives (LOs). During the last month of the lecture period, we introduce a few changes to what is described above. Most notably, the last two tasks of each programming lab are designed to revisit topics covered earlier in the lecture period. During the last lab session, students are presented with the opportunity to solve an old exam.

[1] `https://www.haskell.org/ghc/`

These final few lab sessions are also the part of the course that requires changes when changing the type of examination because here we specifically prepare the students for the exam.

The exam is usually divided into eight main tasks with subtasks, where each of the main tasks is dedicated to a certain topic from the lecture. To ensure we preserve CA while transforming the exam, and to get an overview of what types of task we have, we analyzed the tasks of an exemplary old exam. As a result, we can assign LOs and a task type to each of the tasks. This enables us to make sure that after the transformation to an e-exam the tasks still assess the same LOs, and to find a way to transform each of the task types.

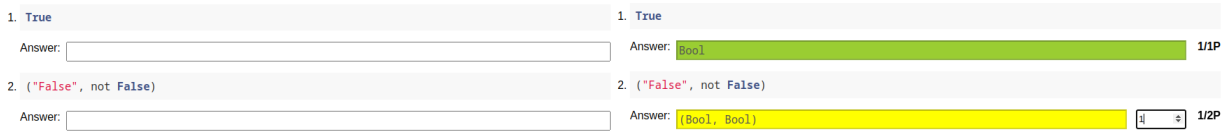
The analysis of the teaching activities showed that the later lab sessions are suited best for familiarizing the students with the e-exam format. Therefore, we decided to move those tasks to YAPS instead of using text editor and terminal as usual. Regarding the exam itself, we found ways to transform each of the task categories we identified to an e-exam. RegExs can be used to detect correct solutions where the solution is a short piece of source code. Yet, a RegEx match is a Boolean decision (i.e., either the given answer matches the RegEx or not), so awarding a reduced amount of points for partially correct solutions is not possible. Tasks that require writing a larger amount of source code can be assessed with randomized property tests via QuickCheck [4] and a new static analysis tool that provides information on used language features and functions. Tasks in which we ask students to prove, e.g., a certain property of a function are a special case though. Here we make use of the proof puzzle task type available in YAPS, with a new evaluation algorithm tailored to our needs. Finally, we address a concern that is not a direct consequence of our analysis, but has been brought up during our discussions: what to do if a student deems a task ambiguous and asks for clarification during the exam? Often, the task is not actually ambiguous, but giving away any information could result in an unfair advantage for the student who asked. Our solution to that is to extend YAPS by integrating a text input field for each task that students can use in any way they want, e.g., for noting how they understood the task.

3 Realization

Taking a more technical perspective, in the following we summarize how we realized the plan derived from the course analysis.

Apart from integrating GHC, to facilitate creating new programming tasks, we devised a template that provides a common (visual) style, layout, and structure, so only task-specific text and code needs to be added using the following workflow: 1. Write the task description into *exercise.html*. This is the page that students see when selecting the task. 2. Write a short `main` function that executes the student code into *main.hs*. Students can execute but not change this code during the exam. 3. Write any task-related code or comments into *functions.hs*. This is the file in which students implement their solutions. It is a good idea to summarize the task in a comment here, so students do not have to switch back and forth between the task description and their solution. 4. Write tests for the student code into *main_test.hs*. 5. Write the code to execute the tests, parse and interpret their output, and report the results back to YAPS into *evaluate.py*. This structure works for almost all tasks without modification, but is still flexible enough to allow for deviations.

Figure 1a shows for two instances of the RegEx task type how they appear in the exam. During the evaluation, the given text input is matched against the specified RegEx and the results are displayed to the examiner as in Fig. 1b. Still, there is one problem with that implementation: what if the correct answer is no answer? Envision a task “Given a certain list comprehension, does it compile, and if so, what is the resulting list?”. We usually design such tasks as a combination of a single choice (for the decision) and



(a) Exam view (b) Evaluation view

Figure 1: Regular expression task

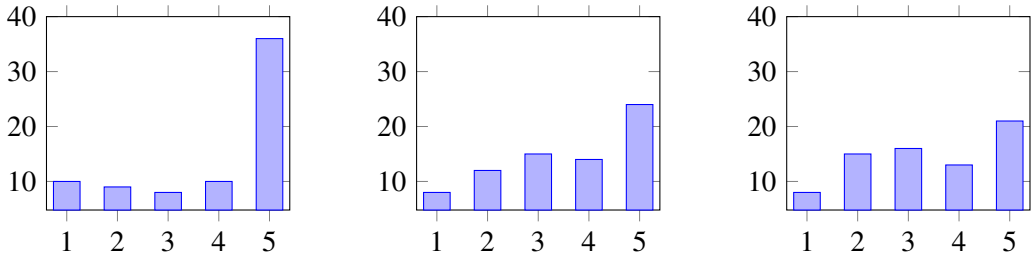
a RegEx task (for the list). If the list comprehension is not valid, the correct answer would be to leave the list input field empty, so we cannot distinguish between "no list" and "no answer." As a remedy, we allow RegEx tasks to optionally depend on a single choice task. The answer to the RegEx part is only evaluated if an answer to the single choice task was given. Additionally, the input field is hidden as soon as the student selects the negative option of the single choice part. This resolves the ambiguity, because now we clearly know whether a task is answered.

In a proof puzzle, students construct their proof from given building blocks. Such a task is defined as follows: First, we list all the available items, and optionally assign a weight to them. Then, we specify possible solutions as sequences of selected items, and assign each solution a number of points. If a student produces one of these solutions exactly, they get the full amount of points. In the original version of this task type, the automatic evaluation is based on the edit distance between the given solution attempt and the specified solution. During testing, this algorithm produced reasonable results. Yet, after conducting the exam, we found that sometimes the resulting points were not congruent with how we manually evaluated these tasks in the past, and that sometimes the results were even unfair. We could not find a different assignment of weights to the items that produces better results, so we decided to devise a new algorithm tailored to our needs. The new algorithm is based on finding correct sequences of items and awards points according to the item weights. There is no subtraction of points, but sequences have clearly-defined entry points (otherwise just using all items in a random order could yield the full amount of points). As the entry points, we use the first items of the predefined solutions.

Many of our programming tasks have certain restrictions, e.g., students must (not) use a certain language feature, or are only allowed to use certain functions. To be able to check these constraints automatically we devised a new tool[2] that itself is written in Haskell. It takes a source code file as input, analyzes it, and outputs certain information on each function, that is defined in the input file, in JavaScript Object Notation (JSON). For each function, the program provides the following information: the name of the function, its arguments, called and locally declared functions, and whether it uses pattern matching/guarded equations/list comprehensions/`case` expressions.

To increase automation during exam creation we developed another tool[3] in Haskell that facilitates writing the RegExs we need to automatically check many of the tasks. It reads a custom, specialized specification language that we call Haskell Task RegEx Specification Language (HTRSL), and outputs JavaScript compatible RegExs (with and without escaped backslashes). For an example, consider the input `("Num" \\ a) "=>" "[" a "]" "->" ["String" | "[" "Char" "]" ]`. It represents the function type `(Num a) => [a] -> String` that can be written in different ways, e.g., omitting the parentheses, writing `[Char]` instead of `String`, or using a different name for the type variable `a`. All of these variations need to be accepted by the generated RegEx. We use the BNF Converter[4] to generate a

[2] `https://collaborating.tuhh.de/cda7728/check-hs-task-restrictions`
[3] `https://collaborating.tuhh.de/cda7728/gen-hs-task-regexs`
[4] `https://hackage.haskell.org/package/BNFC`



(a) Statement 1 (b) Statement 2 (c) Statement 3

Figure 3: Poll results

parser for the language. This way, we obtain an abstract syntax tree that we can traverse to generate the desired RegExs.

4 Evaluation

For evaluating the e-exam we aim to answer the following three questions: What is the degree of automation? Are students satisfied? Are examiners (we) satisfied?

To evaluate the degree of automation of the grading process, we divide all tasks in the e-exam into the following categories: 1. Fully automated: human intervention is not required in any case. 2. Automated if correct: human intervention is only required if the given answer was incorrect (to potentially award partial points) 3. Partially automated: human intervention may even be necessary for correct answers. 4. Not automated: human intervention is required in any case. For an overview, the shares of each category are visualized in Fig. 2.

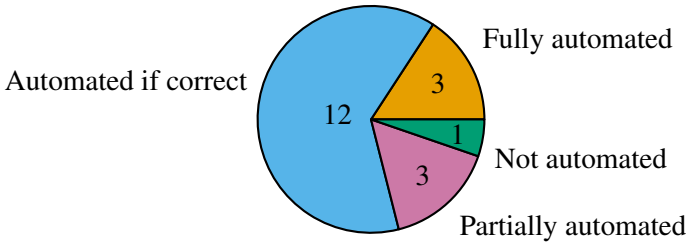


Figure 2: Shares of each automation category

To capture the view of the students, we conducted a short poll immediately after the exam. On the one hand this allowed us to capture the opinion of the students without delay and reduced external influence. On the other hand, we thought that students may be unwilling to fill in an elaborate questionnaire just after finishing an exam. Therefore, we decided to only ask three questions and provide a text input field for additional feedback. The poll read as follows: *Rate the following statements from 1 to 5 (1 = fully disagree, 5 = fully agree) 1. An e-exam is a right format for the lecture "Functional Programming". 2. The programming tasks with compiler support feel like a natural way to answer programming tasks. 3. In an e-exam, I can express my thoughts as good as I could in a paper exam.* Of the 77 exam participants, 73 answered the poll. The results are shown in Fig. 3.

From our own experience, we found that creating an e-exam takes more time than creating a paper exam because in addition to the tasks themselves, we also need to create the automated tests. However, we can confidently expect that this overhead becomes smaller with maturing templates and tools. The technical infrastructure (laptops, network access, etc.) was provided by our university, and we did not

have to concern ourselves with printing the exams, making sure they are complete, and distributing and collecting them during the exam, so the process of conducting the exam is easier now. Furthermore, the time required for grading is also reduced, because in most cases only tasks that were answered incorrectly need to be examined.

5 Summary & Future Work

We showed how a traditional paper exam with complex proof and programming tasks can be transformed to an e-exam with automated grading, but without sacrificing CA. Through careful analysis of the course and previous exams we can ensure that exam quality does not degrade during the transformation. For realizing the e-exam we built upon existing software that we extended with a Haskell compiler, tasks that can be checked with RegExs, a new algorithm to automatically evaluate proofs, and a general purpose comment field for students. Additionally, we introduced two new tools that substantially support automation: one that analyzes student code for task-relevant features, and one that generates suitable RegExs from a more high-level description language. We achieved that almost all tasks can be graded automatically at least in part, and students as well as examiners are largely satisfied with the resulting e-exam.

In the future we want to investigate how a more fine-grained automated grading can be achieved because currently awarding reduced amounts of points for partial solutions is often not possible (at least not as reliably as we require it for an exam). Moreover, because creating an e-exam requires more effort than creating a paper exam, we also want to explore how this process can be automated further. Our vision is that most of the e-exam can be generated automatically from a single document that contains the task texts, solutions, and point distribution.

References

[1] Fin Hendrik Bahnsen & Goerschwin Fey (2021): *YAPS - Your Open Examination System for Activating and emPowering Students*. In: *2021 16th ICCSE*, pp. 98–103, doi:10.1109/ICCSE51940.2021.9569549.

[2] Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Michael Sperber, Marcus Crestani, Herbert Klaeren & Eric Knauel (2008): *Htdp and Dmda in the Battlefield: A Case Study in First-Year Programming Instruction*. In: *Proc. of the 2008 FDPE*, pp. 1–12, doi:10.1145/1411260.1411262.

[3] John Biggs (1996): *Enhancing Teaching through Constructive Alignment*. *Higher Education* 32(3), pp. 347–364, doi:10.1007/BF00138871.

[4] Koen Claessen & John Hughes (2000): *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In: *Proc. of the 5th ACM SIGPLAN ICFP*, pp. 268–279, doi:10.1145/351240.351266.

[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2004): *The TeachScheme! Project: Computing and Programming for Every Student*. *Computer Science Education* 14(1), pp. 55–77, doi:10.1076/csed.14.1.55.23499.

[6] David R. Krathwohl (2002): *A Revision of Bloom's Taxonomy: An Overview*. *Theory Into Practice* 41(4), pp. 212–218, doi:10.1207/s15430421tip4104_2.

[7] Daniel Sitzmann, Karsten Kruse, Dennis Gallaun, Norwin Kubick, Björn Reinhold, Manuel Schnabel, Lars Thoms, Helena Barbas & Sina Meiling (2022): *Aufbau eines mobilen Testcenters für die Hamburger Hochschulen im Rahmen des Projekts MINTFIT E-Assessment*. *Die Hochschullehre* 8, pp. 113–129, doi:10.3278/HSL2208W.