EPTCS 382

Proceedings of the

Twelfth International Workshop on Trends in Functional Programming in Education

Boston, Massachusetts, USA, 12th January 2023

Edited by: Elena Machkasova

Published: 14th August 2023
DOI: 10.4204/EPTCS.382
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii

ProofBuddy: A Proof Assistant for Learning and Monitoring 1
Nadine Karsten, Frederik Krogsdal Jacobsen, Kim Jana Eiken, Uwe Nestmann and Jørgen Villadsen

Computer Aided Design and Grading for an Electronic Functional Programming Exam 22
Ole Lübke, Konrad Fuger, Fin Hendrik Bahnsen, Katrin Billerbeck and Sibylle Schupp

Regular Expressions in a CS Formal Languages Course 45
Marco T. Morazán

Disco: A Functional Programming Language for Discrete Mathematics 64
Brent A. Yorgey

Preface

This volume contains proceedings from the 12th International Workshop on Trends in Functional Programming in Education (TFPIE). The workshop was held at UMass Boston, Boston, Massachusetts, USA on January 12th 2023.

TFPIE series

The goal of TFPIE is to gather researchers, professors, teachers, and all professionals interested in functional programming in education. This includes the teaching of functional programming, but also the application of functional programming as a tool for teaching other topics, e.g. computational concepts, complexity, logic and reasoning, and even disciplines, e.g. philosophy or music. TFPIE is the heir of previous events, like Functional and Declarative Programming in Education (FDPE), to which it owes a great deal and from which it has borrowed experience and ideas. TFPIE workshops have previously been held in St Andrews, Scotland (2012), Provo Utah, USA (2013), Soesterberg, The Netherlands (2014), Sophia-Antipolis, France (2015), College Park, USA (2016), Canterbury, UK (2017), Gothenburg (2018) and Vancouver (2019), Krakow, Poland (2020) and online due to COVID-19 (2021, 2022, with some talks from TFPIE 2022 also presented in person at the Lambda Days in Krakow, Poland).

Programme and keynotes

TFPIE 2023 was honored to feature a keynote "How to Plan Programs" by Shriram Krishnamurthi. The sessions speakers were Brent Yorgey, Nadine Karsten, Frederik Krogsdal Jacobsen, Marco T. Morazán, Enzo Alda, Ole Lübke, Rui Barata, and Simão Melo-de-Sousa.

Submissions

TFPIE 2023 received, and accepted, six submissions for presentation at the workshop. The post-workshop review process received five submissions, which were reviewed by the program committee, assuming scientific journal standards of publication. Four articles were selected for publication as the result of this process.

Table of contents

1. Nadine Karsten, Frederik Krogsdal Jacobsen, Kim Jana Eiken, Uwe Nestmann and Jørgen Villadsen "ProofBuddy: A Proof Assistant for Learning and Monitorin"
2. Ole Lübke, Konrad Fuger, Fin Hendrik Bahnsen, Katrin Billerbeck and Sibylle Schupp "Computer Aided Design and Grading for an Electronic Functional Programming Exam"
3. Marco T. Morazán "Regular Expressions in a CS Formal Languages Course"
4. Brent Yorgey "Disco: A Functional Programming Language for Discrete Mathematics"

Elena Machkasova (Ed.): International Workshop on Trends
in Functional Programming in Education 2023 (TFPIE 2023)
EPTCS 382, 2023, pp. ii–iii, doi:10.4204/EPTCS.382.0

© Elena Machkasova
This work is licensed under the
Creative Commons Attribution License.

Concluding remarks

Many thanks to Stephen Chang (TFP 2023 chair) who organized an excellent in-person conference at UMass Boston and handled all of the related logistics. Finally, we would like to thank the members of the programme committee

- Christopher Anand - McMaster University, Canada
- Attila Egri-Nagy - Akita International University, Japan
- Jason Hemann - Seton Hall University, USA
- Kevin Kappelmann - Technical University of Munich, Germany
- Elena Machkasova (Chair) - University of Minnesota Morris, USA
- Kristina Sojakova - INRIA, France
- Jørgen Villadsen - Technical University of Denmark, Denmark

who worked very hard to provide the authors feedback for improving their papers. We hope that you enjoy reading the results!

PROOFBUDDY:
A Proof Assistant for Learning and Monitoring

Nadine Karsten
Technische Universität Berlin
Berlin, Germany
n.karsten@tu-berlin.de

Frederik Krogsdal Jacobsen
Technical University of Denmark
Kongens Lyngby, Denmark
fkjac@dtu.dk

Kim Jana Eiken
Technische Universität Berlin
Berlin, Germany
k.eiken@campus.tu-berlin.de

Uwe Nestmann
Technische Universität Berlin
Berlin, Germany
uwe.nestmann@tu-berlin.de

Jørgen Villadsen
Technical University of Denmark
Kongens Lyngby, Denmark
jovi@dtu.dk

Proof competence, i.e. the ability to write and check (mathematical) proofs, is an important skill in Computer Science, but for many students it represents a difficult challenge. The main issues are the correct use of formal language and the ascertainment of whether proofs, especially the students' own, are complete and correct. Many authors have suggested using proof assistants to assist in teaching proof competence, but the efficacy of the approach is unclear. To improve the state of affairs, we introduce PROOFBUDDY: a web-based tool using the Isabelle proof assistant which enables researchers to conduct studies of the efficacy of approaches to using proof assistants in education by collecting fine-grained data about the way students interact with proof assistants. We have performed a preliminary usability study of PROOFBUDDY at the Technical University of Denmark.

1 Introduction

The curriculum of a Bachelor study programme in Computer Science typically includes programming skills, technical understanding about computers, theoretical knowledge and math. Especially the latter two cause difficulties for students. Theoretical courses cover formal languages, automata, logic, complexity and computability, which all have in common that they build on mathematical structures and proofs about them. Hence Computer Science students have to write and verify proofs in several theoretical courses. This requires *proof competence*, which includes the following four specific competencies [7]: (1) *professional* competence describes the contextual knowledge about the proposition that has to be proved; (2) *representation* competence is the ability to write down a proof in sufficiently formal language; (3) *communication* competence is then understood as arguing about the procedure and solution of a proof; (4) finally, *methodological* competence summarizes three aspects [16]: proof scheme, proof structure and chain of conclusions.

All of these competencies are taught as part of introductory courses in Theoretical Computer Science, but proof competence with the above-mentioned facets is usually not an explicit part of the curriculum. As a result, students work through proofs that the teacher presents during lectures and try to replicate [33]. Most students fail this way. In [14], Frede and Knobelsdorf analyzed the homework and the written exams of an introductory course at Technische Universität Berlin (TUB). The result was that students make the most mistakes in exercises with proofs, regardless of the final grade. The main challenges in writing proofs are the usage of formal language while writing proofs and the ascertainment of whether a proof is complete and correct [19–21]. When students try to write proofs in introductory courses in Theoretical Computer Science, the failure rate is therefore high.

Elena Machkasova (Ed.): International Workshop on Trends in Functional Programming in Education 2023 (TFPIE 2023)
EPTCS 382, 2023, pp. 1–21, doi:10.4204/EPTCS.382.1

© N. Karsten, F. K. Jacobsen, K. J. Eiken, U. Nestmann & J. Villadsen
This work is licensed under the Creative Commons Attribution License.

As we will see in the next section, proof assistants have often been suggested as a tool to improve proof competence. A proof assistant can be thought of as a functional programming language combined with a language for reasoning about programs. This allows users of proof assistants to write proofs in a formal, structured way, and get instant feedback on the correctness of their proofs from the proof checker. Proponents of using proof assistants in education claim many alleged benefits of the approach, but unfortunately, these claims are rarely supported by more than anecdotal evidence or surveys asking students to report their attitudes about the efficacy of the approach. Conversely, there are reasonable doubts about the efficacy of the approach: it is not clear that proof competences gained using proof assistants transfer to pen-and-paper proofs, proof assistants may enable a certain flavor of "brute-force proving" and introducing students to a proof assistant may use precious teaching time that could be better spent elsewhere.

Clearly, authors should do something to test these claims, but as we will see in the next section this is rarely done, at least partially due to a lack of tools that would enable the studies needed to test them. Inspired by similar approaches in studying the efficacy of didactic techniques when teaching functional programming and mathematics, we introduce PROOFBUDDY: an instrumented version of the Isabelle proof assistant [27, 28]. The idea of PROOFBUDDY is to collect fine-grained data about the interactions of students with the proof assistant and to use this data to conduct studies of the efficacy of various didactic approaches. Additionally, PROOFBUDDY has optional features to attempt to counteract some of the common doubts about the use of proof assistants in education. We have evaluated the usability of PROOFBUDDY and the data collection capabilities based on a set of research questions that we imagine future didactic studies could want answers to and found that the type of data collected by the tool is sufficient to answer these types of questions.

In summary, our contributions are:

- An overview of the current issues in studying the efficacy of using proof assistants in education.
- PROOFBUDDY: a version of the Isabelle proof assistant with instrumentation for collecting data about user interaction with the proof assistant.
- A preliminary evaluation of the usability of PROOFBUDDY from the perspective of students using the tool.
- A preliminary evaluation of the adequacy of the interaction data collected by PROOFBUDDY for conducting didactic studies.

In the next section, we will discuss existing uses of proof assistants in education, including the claimed benefits and drawbacks of using proof assistants in education, as well as some of the tools from the domain of functional programming that have inspired the design of PROOFBUDDY. We will outline potential approaches to mitigate the drawbacks and investigate the claimed benefits of using proof assistants in education in Section 3. Next, we describe the features (inspired by these potential approaches) and implementation of PROOFBUDDY in Section 4. In Section 5 we evaluate PROOFBUDDY from two perspectives: the adequacy of the collected data for conducting didactic studies and the usability from the perspective of students. Finally we outline future work in Section 6 before concluding in Section 7.

2 Related Work

This section covers work related to PROOFBUDDY, by which we mean not only descriptions of superficially similar tools, but also the articles and studies that have motivated the development of PROOFBUDDY, and the tools from similar fields that have inspired its design. We will cover a selection of reports on

using proof assistants in education, then summarize the claimed benefits and observed drawbacks of doing so. Finally, we will note some approaches from similar fields which have inspired the design of PROOFBUDDY.

2.1 Using Proof Assistants in Education

The idea to use proof assistants in education is not new, but the scientific results of these efforts have mostly been experience reports, and not rigorous evaluations of the efficacy of the approach. In this paper we will give only a short overview of approaches. Several courses target students in higher semesters and assume proof competence and functional programming skills [26, 36]. Knobelsdorf et al. [21] report on the use of the proof assistant Coq [3] to enhance proof competence, with the following result: while students were able to prove theorems with Coq after the course, their ability to write pen-and-paper proofs was not significantly better than before the course. Böhne and Kreitz [5] therefore attempt to improve pen-and-paper proof competences by requiring students to add comments in the formal Coq proof; nevertheless the improvement in writing pen-and-paper proofs was only minimal.

Some have also tried to integrate a proof assistant in first-year courses. Avigad [2] used the proof assistant Lean [25] in a logic course in 2015 to combine mathematical language and natural language in proofs. Avigad describes only anecdotal evidence and concludes that it seems necessary to develop quantitative methods and tools for data collection before carrying out a comparative study of the efficacy of the approach. Thoma and Iannone [39] describe a study where students could choose to use Lean in addition to the usual material in a first-year university course. The small number of students who used Lean in voluntary workshops used more mathematical language during an interview and structured their proofs better. Because Lean was not mandatory it is possible that only the students who were already doing well in the course chose to also use the proof assistant. Lean was also used for the Natural Number Game [9], which is a web application designed to teach formal proofs in a proof assistant using basic theorems in arithmetic and logic. Kreitz, Knobelsdorf and Böhne [4] consider a first-semester course where Coq is used in the lectures and the exercises to support the handling of formalisms, but this course was not realized. The web tool Waterproof by Wemmenhoven et al. [43] allows users to write Coq proofs with natural language. The authors developed a library to teach students in Analysis I where using the proof assistant was voluntary. The students who used the tool had better grades at the end of the course, but again this may simply be a result of self-selection bias. At TUB, we integrated Isabelle in a second semester course in 2021. We used Isabelle to introduce propositional and first-order logic step by step, but using Isabelle for the exercises and homework was optional. The students had fun in proving but it was time-consuming for them to step into Isabelle.

Lurch was a “proof-checking word processor” in which students could write natural language texts and mark certain parts of their input as having mathematical content, which the tool would then check [10, 11]. The tool focused on good user experience for beginners, and the expressivity of the formalism is unclear. The desktop version of the tool is no longer available, and the development of a new web-based version seems to have ceased in 2018 [12]. The web tool Carnap [22] offers a wide range of exercises in mathematics and logic. In 2017 there were several courses where different approaches with different students where tested. The author anecdotally found that Carnap offered exercises appropriate for different levels of competence. The AProS project is an effort to develop a number of courses and tools to teach logic and proofs [34]. These courses are computer-taught, not only computer-assisted, and include a number of interactive environments which are essentially proof assistants. Burke provides anecdotal evidence for the efficacy of the AProS project [8]. Clide [23, 30] is a web application based on Isabelle. Clide was implemented already in 2013 and consists of an editor integrated within a web interface, and an

Isabelle backend for checking proofs. Clide was not focused on learning, however, but focused on allowing collaborative writing and editing of proofs. The Clide application seems to no longer be available.

Many purpose-built proof assistants designed for learning specific topics have been developed, including Jape [37, 38], ProofWeb [17], SPA [31], SeCaV [15] and NaDeA [40–42].

The second and fifth author have previously conducted a study of student experiences using Isabelle, but our methods suffered from a lack of data on actual student behavior and interactions when using the proof assistant [18]. Knobelsdorf et al. limited themselves to manually observing student behavior and categorizing the types of questions asked by students [21]. Mariotti has conducted a number of studies on learning to prove with a dynamic geometry environment, but similarly describes only manual observations and interviews with students [24].

2.2 Claimed Benefits and Drawbacks of Using Proof Assistants in Education

As described in the previous sections, there have been many attempts to use proof assistants in education. In this section, we will summarize the claimed benefits and drawbacks of this approach in the literature.

We start with the claimed benefits of using proof assistants in education, of which there are many:

- Proof assistants are useful for teaching mathematics [2, 4, 10, 11, 17]
- Proof assistants are useful for teaching functional programming [4, 5, 17, 18, 21, 26]
- Proof assistants are useful for teaching logic [2, 4, 5, 8, 15, 17, 18, 21, 26, 34, 36, 41]
- Proof assistants are useful for teaching abstract thinking [4, 11, 34, 36]
- Proof assistants make the rules of formal reasoning clear [2, 4, 11]
- Proof assistants help students learn how to structure proofs [4, 5, 8, 10, 11, 17, 19, 21, 31]
- Students are helped by instant feedback on their proofs [2, 4, 8, 10–12, 15, 21, 26]
- Proof competence gained using a proof assistant transfers to pen-and-paper [2, 12, 26, 34]
- Proof assistants help students fix their proof errors as early as possible [4, 11]
- Proof assistants make it easier for students to get started on a proof [4, 15]
- Students consult formal definitions to gain understanding [18, 42]
- Students experiment with formal definitions to gain understanding [18, 26]
- Proof assistants help students experiment with proof ideas [4]
- Correcting assignments checked by proof assistants is easier [8]

Some authors have however also observed various difficulties that students encounter when trying to use proof assistants, including:

- Difficulties learning proof assistant syntax [2, 12]
- Difficulties understanding proof assistant error messages [2]
- Difficulties transferring proof competencies from proof assistants to pen-and-paper [5, 21]
- Difficulties stating properties formally [8]
- Technical difficulties in installing and using a proof assistant [8, 17, 21]
- Difficulties understanding the very expressive language of a proof assistant [17, 21]

- Difficulties remembering formal proof rules [21]

Some authors also note potential issues of using a proof assistant in education from the course instructor's point of view, including:

- The overhead of introducing a proof assistant to students [2, 26]
- The need to develop specialized proof scripts for each topic [4]
- The need to design exercises that cannot be solved by brute force [5, 11, 17, 21, 41]
- The worry that electronic exams using proof assistants may make it easier to cheat [17, 26]
- That proof assistants have automation that makes too many exercises trivial [17]

Finally, several authors note methodological issues in attempting to provide evidence for any of the above-mentioned claims:

- No suitable quantitative measures of proof competence exist [2]
- It is difficult to collect evidence about how students interact with proof assistants [5, 18]
- It is unclear how to compare the efficacy of approaches based on proof assistants with approaches based on pen-and-paper approaches in a fair way [18, 26]

2.3 Approaches From Similar Fields

Wrenn and Krishnamurthi have developed a tool to enable problem comprehension in functional programming by letting students test their understanding by writing property-based tests against a specification before writing code [46], and shown that students will use the tool voluntarily [47]. In studies using this tool, they found that students still had several types of misunderstandings that the tool or the lectures could have been improved to alleviate [45] and that course instructors had blind spots in predicting the misunderstandings students had [29]. Instrumenting the tool with data collection facilities allowed them to inspect student interactions with the tool in a fine-grained manner, and the tool thus helped instructors discover students' actual misunderstandings, which were different from what the instructors had imagined.

Aleven and Koedinger have designed a so-called Cognitive Tutor, which is a specially designed "proof assistant" instrumented with features for tracking student behavior and guiding students to explain their work [1]. They show that the use of this tool with the guidance features enabled improves student learning outcomes in geometry. The study was enabled by instrumenting the Cognitive Tutor with facilities for collecting fine-grained data about time spent and steps taken in the exercises completed by students using the tool.

3 Enabling Educational Research on Proof Assistants

As mentioned in the previous section, there are few rigorous studies about the influence of proof assistants on the learning of proof competences, but many claims about their efficacy. In this section we will discuss the importance of performing studies on the impact of proof assistants in education and the obstacles that we need to overcome to enable these studies. First, however, we discuss approaches to curtailing the observed difficulties students and instructors encounter when using proof assistants in education.

3.1 Making it Easier to Learn Proof Assistants

Proof assistants are generally not developed for students, but for expert users, and the learning curve is thus very steep. For beginning students, using any formal language can be a big challenge, and the rigorous languages of proof assistants can be even more difficult to break into. Instructors could potentially decrease the challenge by introducing the language one element at a time and developing tools that can guide the students for the specific learning goals of each activity. One could imagine this approach combined with adaptive learning tools to guide students at their own pace. A related issue is that proof assistants typically require users to remember not only the contents, but also the formal names of proof rules. Course developers could curtail this issue by providing an easy way to see the relevant proof rules in each learning activity without having to look them up in a larger list containing many irrelevant rules. Students can then engage with the content of the course instead of memorizing names.

Another issue is that students find error messages from proof assistants vague and confusing. This is typically a result of the language of the proof assistant being so expressive, and implementations being so general, that errors refer to concepts that students are not familiar with, e.g. type classes or functors. In many educational contexts, the full expressive power of the proof assistant is not necessary, and instructors could potentially flatten the learning curve by restricting the language to the fragment required by each activity. This would allow more specific error messages and perhaps even hints about why a specific application of a proof rule is wrong.

Some students have issues installing and using proof assistants from a purely technical perspective. Some of the issues could be reduced by developing web-based tools, which have the additional benefit of being easy to update such that students are always on the newest version of the tool in case any issues with the instrumentation are found during the course.

Finally, students have issues moving between proof assistants and traditional pen-and-paper proofs and statements. This occurs both when formalizing properties stated in natural language and when transferring the proof competences developed with a proof assistant to competences in writing pen-and-paper proofs. We are not convinced that it is possible to solve these issues by technical means, but approaches where the difference between proof assistant and pen-and-paper is gradually minimized seem promising.

3.2 Making it Easier to Use Proof Assistants in Education

Choosing to use a proof assistant in education unavoidably introduces the need to balance the time spent learning the actual course content and the time spent learning to use the proof assistant. The challenge is to design a course that uses the proof assistant as a tool instead of a course which teaches how to use proof assistants [26] (though a course which aims to do this with intention can of course also be valuable).

When designing exercises which are intended to be solved using proof assistants, there are several potential issues: exercises may be too easy to solve by brute force, automation in the proof assistant may make the exercises trivial and using electronic exercises for assignments or exams may make it easier for students to cheat. Restricting the language of the proof assistant (as described in the previous section) may also facilitate the design of exercises that are difficult to brute force and remove excessive automation.

The problem of designing exercises and supporting developments for each learning activity remains. We conjecture that this problem is mainly historical, i.e. caused by the fact that many courses have pen-and-paper exercises developed through several years. When designing new learning activities, we conjecture that using a proof assistant may actually expedite the development of good exercises since obscure or easily overlooked corner cases must be handled during the development and are thus not present to confuse students during the actual course.

3.3 Gathering Evidence

One of the main issues with implementing proof assistants in education is that there is little evidence of the efficacy of the approach in improving student proof competences. It is thus unclear whether spending time and resources on designing course material using proof assistants will actually result in students learning more. A compounding issue is that it is unclear how to measure proof competence across approaches based on proof assistants and approaches based on pen-and-paper in a quantitative way and without favoring one of the approaches. When considering studies, instructors must thus be careful to specify the learning objectives they are measuring precisely: is the intention of their course to improve proof competency *in general* or for instance specifically in pen-and-paper proofs?

The other major obstacle to conducting studies of proof assistants in education is that collecting objective, quantitative data about student interactions with proof assistants and learning outcomes is difficult. Most reports thus rely on surveys which ask students to self-report about their experiences or vague measures based on average grades in the course. One of the main objectives of PROOFBUDDY is to enable researchers to conduct studies of the efficacy of approaches to using proof assistants in education by collecting fine-grained data about the way students interact with proof assistants.

4 PROOFBUDDY

Inspired by the claimed benefits and drawbacks of using proof assistants for educational purposes described in the previous section, we have developed PROOFBUDDY: an instrumented version of the Isabelle proof assistant which is accessible through a web interface. The tool communicates with a full Isabelle proof assistant running on our server to check proofs and programs and additionally collects data about how users interact with the tool. It would also have been possible to instrument one of the usual Isabelle editors Isabelle/jEdit or Isabelle/VSCode. However, that would mean having to update the tool with every new Isabelle version, whereas communicating via the Isabelle server protocol allows us to develop against a more stable target. Besides the advantage that users do not have to install a special version of Isabelle, a web tool gives us the opportunity to add new features directly on the server without having users reinstall the tool. Through a management backend on the server it is possible to organize different courses, teacher and students and the collection of data (log-files and the submitted Isabelle theories). Furthermore a web interface yields more flexibility, and it is conceivable to add other languages like Coq or Lean while preserving a unified interface. Figure 1 shows a screenshot of PROOFBUDDY.

PROOFBUDDY is a web application and hence divided into two parts: the frontend, running in the browser, and the backend, running on the server. Figure 2 outlines the system architecture of PROOFBUDDY. The design decision to use Isabelle in the backend instead of the frontend came as a result of Isabelle's development languages, ML and Scala, which are difficult to run in a browser. Furthermore it is easier to log the communication with the Isabelle server in the backend and save the theories which are sent to be verified. The frontend and backend of our application communicate via Socket.IO [35], allowing for instantaneous and bi-directional data transfer. Socket.IO is an event-driven library based on the WebSocket protocol [13]. In browsers where WebSockets are not supported, Socket.IO assures a stable connection via HTTP long-polling.

4.1 Frontend

The user interface of PROOFBUDDY mainly consists of three panes: an editor pane, an output pane and a PDF viewer. The *editor pane* is the main interactive component. An Isar proof of the "drinker's principle"

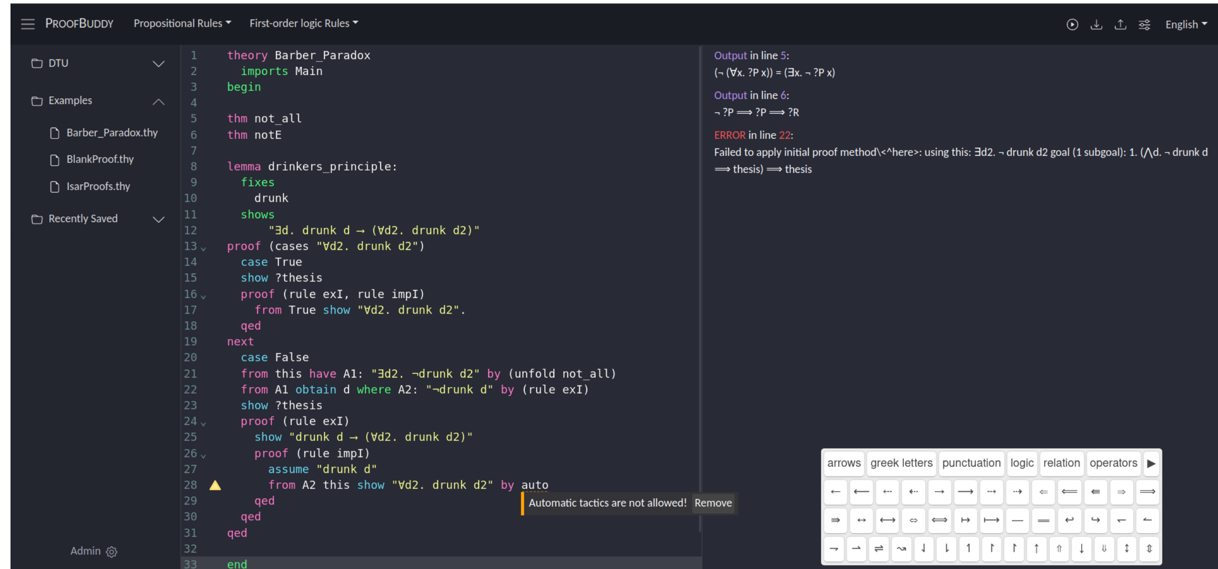


Figure 1: Screenshot of the interface of PROOFBUDDY with file browser on the left, editor in the middle and output messages on the right. Note also the linter popup (bottom center) and the symbol keyboard.

is shown in the *editor pane* in the middle of Figure 1.

PROOFBUDDY includes a parser for the language of Isabelle which enables syntax highlighting, autocompletion, code folding, bracket closing, search and replace functionality, etc. The editor of PROOFBUDDY thus functions like a typical integrated development environment (IDE). Since Isabelle is solely running in the backend, PROOFBUDDY does not have the ability of Isabelle/jEdit and Isabelle/VSCode to look up definitions and display type information by clicking on names of e.g. functions and theorems.

Instructors can restrict the language, and thus expressivity, of Isabelle on an activity-to-activity basis by adding linters, which disallow certain syntactic constructs. Linters display errors and warnings instantly in form of popups directly at the point where the failure occurs. We have implemented a linter based on regular expressions (as shown in Figure 1 and Figure 3) to warn about the usage of the automatic tactics auto, simp, arith and blast. At this point of development, the linters do not prevent users from asking Isabelle to check their work even if a linter detects the usage of prohibited syntax for the activity at hand. We could also restrict syntax by hiding definitions via a prelude, but this would not give users any specific information about their mistake when attempting to use e.g. a prohibited tactic, since the error from Isabelle would simply be about attempting to use an undefined tactic. Students can currently toggle the linter for automatic tactics through a switch on the user interface, but this feature can be removed to force students to only use automation in specific activities. It is thus possible to introduce features one at a time, ensuring that students can only use the features that they are supposed to learn in a given activity.

The *output pane*, located on the right-hand side, displays the feedback from Isabelle regarding the correctness of the current development. The closable *PDF viewer* allows the display of tutorials, lecture notes, exercise descriptions, etc. within the tool.

The collapsible *sidebar* contains a file browser and the profile of the current user. Users must log in to access PROOFBUDDY. User profiles are used to store progress and access previously created theories as well as to track interactions of individual users in the collected data. We did however implement a guest login to allow testing of PROOFBUDDY. The *toolbar* contains buttons to run the development (i.e.

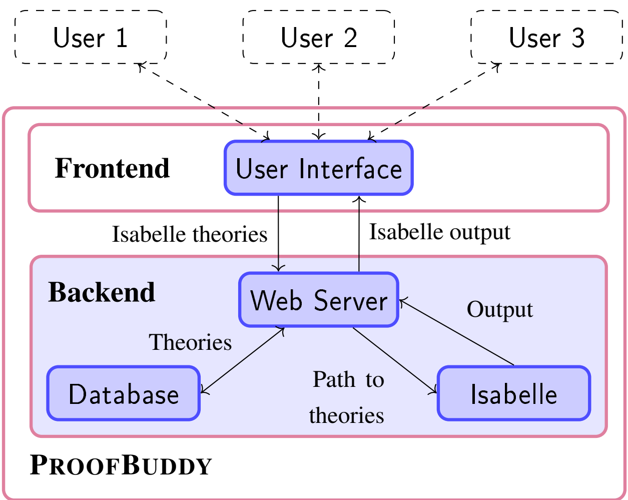


Figure 2: Architecture of ProofBuddy. Note that each user has their own instance of the frontend running in their browser, while there is only a single backend.

send the content of the editor to Isabelle for checking) and to download and upload Isabelle theories. Additionally, the toolbar contains dropdown menus which list the names and definitions of relevant proof rules on an activity-to-activity basis, as shown in Figure 3. Users can insert rule names at the current cursor position in the editor by clicking on a rule in the list.

Users can open the *symbol keyboard* by clicking the keyboard-button in the bottom right corner as seen in Figure 3. As shown in Figure 1, the keyboard offers an easy way to enter relevant mathematical and logical symbols, which instructors can configure on an activity-to-activity basis.

4.2 Backend

The backend of ProofBuddy is responsible for handling user authentication and data management, but its main functionality is to check the correctness of received developments using Isabelle.

The Isabelle proof assistant usually runs as two processes: a daemon, the Isabelle server, and an interactive application, the Isabelle client, through which theories can be sent to be checked by the server [44]. The Isabelle server listens on a TCP socket and allows for bi-directional communication with multiple clients following a protocol of structured messages. The ProofBuddy backend communicates with Isabelle via the Isabelle client by sending requests to check the developments it receives (checking is incremental as in the usual Isabelle development environments, so only theories with changes are checked). The messages consist of the session ID and a number of theories. The Isabelle server acknowledges the request and answers asynchronously whenever it finishes a task. The message of the Isabelle server includes all the usual error messages which are known from Isabelle/jEdit. Additionally, the Isabelle server periodically sends progress reports on ongoing tasks. The ProofBuddy backend starts the Isabelle client as a child process after ensuring that the Isabelle server is running. By writing to the input stream and reading the output stream of the Isabelle client, the backend is able to manage Isabelle sessions and check theories using Isabelle while simultaneously logging the interactions.

Isabelle sessions [44] consist of a collection of related Isabelle theories and build upon other Isabelle sessions. It is necessary to add Isabelle theories to an Isabelle session in order to check them using the

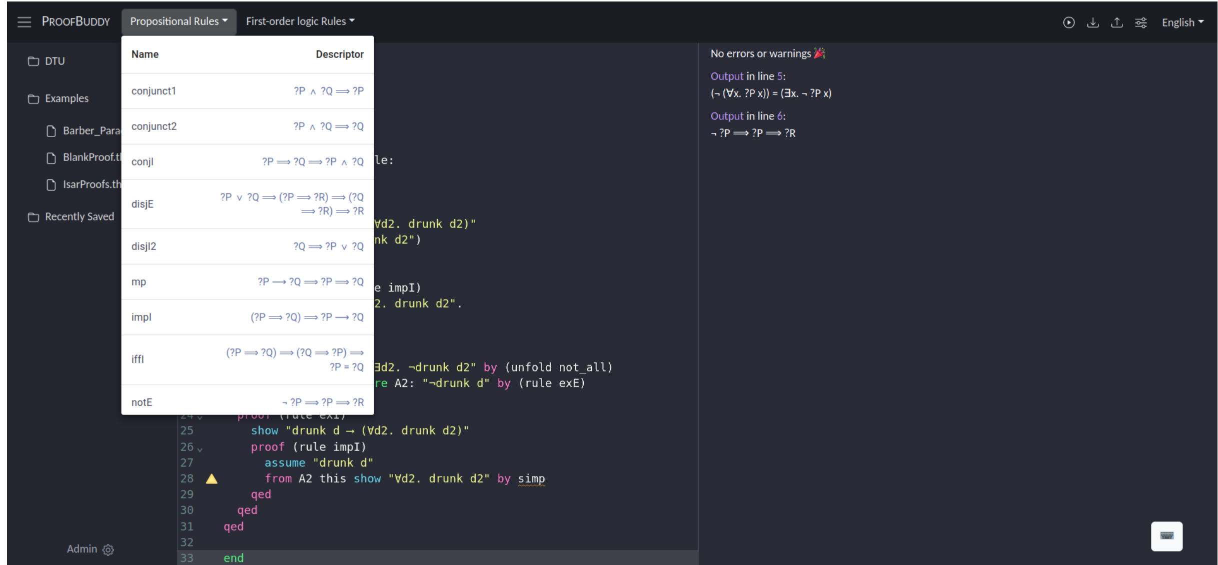


Figure 3: Screenshot of the interface of PROOFBUDDY with a reference list of proof rules. Note the yellow warning triangle and yellow underline of the "simp" tactic from the linter.

Isabelle server. The current version of PROOFBUDDY always uses Isabelle/HOL as the parent session. Hence, PROOFBUDDY makes the entire functionality of Isabelle/HOL available. For PROOFBUDDY (and thus Isabelle) to distinguish theories with the same name that are written by different users, each user has an individual Isabelle session. Adding theories to an Isabelle session automatically results in Isabelle checking the correctness and completeness of the theories. Once the Isabelle server has checked a development, the web server sends feedback from Isabelle directly to the frontend using Socket.IO, where it is formatted and displayed to the user.

4.3 Advantages of PROOFBUDDY

We designed PROOFBUDDY in response to the issues described in Section 3, and we will here highlight some of the ways in which PROOFBUDDY provides opportunities to alleviate these issues.

Due to the design decision of implementing a web interface, students do not need to install anything, and instructors can be certain that all students are using the same version of the tool.

The parser and the linters offer the possibility to restrict the input language used in PROOFBUDDY. The restriction of the input language makes it possible to introduce features one at a time, ensuring that students can only use the features that they are supposed to learn about in a given activity. The dropdown menus containing reminders of proof rules relevant to the activity at hand could also make it easier for students to engage with the content of the course by eliminating time spent looking up proof rules. Restricting the input language additionally could make it easier to design exercises that are harder to solve by brute force, e.g. by disallowing automation or certain proof rules. Restricting the input language could also be used to force students to write proofs in a style that resembles pen-and-paper proofs.

By formatting and specializing the messages from Isabelle in the frontend before displaying them, it is possible to give better feedback as described in Section 3. This would make it possible to add more, and also more useful, feedback to the output, depending on the failure or the previous behavior of the user.

Instructors can develop an interactive tutorial which forces students to complete exercises in a certain

order by structuring the activities in ProofBuddy using the built-in file browser and PDF viewer. This also allows for tailoring exercises to the pace and needs of the individual student.

The major benefit of ProofBuddy is the ability to monitor the progress and behavior of each student as they learn to program and prove in Isabelle/HOL. Therefore we are able to collect data to evaluate the behavior of students. This data consists of the contents of student theories saved whenever the student sends the theory for checking, annotated with time stamps. Monitoring the individual keystrokes and cursor position could also be possible to get even more detail. A high frequency of checking a theory can indicate that the student tries something without really thinking about why a proof is not accepted. If many students struggle with the same exercise, the instructor can improve the exercises or give a better introduction. When analyzing the failure it is also possible to adapt the next exercises. Using ProofBuddy offers two opportunities: (1) educational studies about the impact of proving with proof assistants can be performed with the help of the collected data and (2) learning analytics to improve and adapt the exercises to each student depending on the failures. In the next section, we will evaluate the extent to which ProofBuddy is useful for the first of these opportunities.

5 Evaluation of ProofBuddy

We evaluate ProofBuddy both from the perspective of researchers wanting to carry out didactic studies using data collected by the tool and from the perspective of students using the tool. Since we are not aware of any rigorous studies of how proof assistants impact learning, we are unable to test the adequacy of ProofBuddy with regards to concrete research questions from the literature, but instead evaluate ProofBuddy from the perspective of researchers by testing the adequacy of the data collected by ProofBuddy for answering research questions that we imagine could be part of future didactic studies. We note that ProofBuddy can of course only be used to answer questions about the behavior of students when interacting with a proof assistant.

Any comparative study between approaches based on proof assistants and approaches based on pen-and-paper would still need additional work to answer questions about the behavior of students when writing proofs on paper. Note also that this evaluation is not itself a didactic study, but simply an evaluation of the usefulness of the ProofBuddy tool for conducting future studies. The evaluation from the perspective of students concerns the usability of ProofBuddy, both in general and as compared to the standard editors distributed with Isabelle.

5.1 Didactic Context

We carried out the evaluation of ProofBuddy as part of the existing MSc-level course *Automated Reasoning* at the Technical University of Denmark (DTU) in the spring of 2023. This is an elective course for advanced Computer Science students, and the main topic is learning the use and theory of Isabelle. We thus expect students following the course to already be proficient at pen-and-paper proofs, and the main purpose of evaluating ProofBuddy using a population of students following the course is not to teach the students, but to collect data about how students interact with ProofBuddy. Table 1 contains a brief overview of the course plan and more information about the course can be found at `https://kurser.dtu.dk/course/2022-2023/02256`. We carried out the evaluation in week 7 of the course, at which point the students were already somewhat familiar with the use of Isabelle.

Weeks	Topics
1 to 2	Introduction, programming and proving, sequent calculus
3 to 4	Logic and proof beyond equality, natural deduction (first-order logic)
5 to 6	Isar: a language for structured proofs, natural deduction (higher-order logic)
7 to 9	Simple type theory
10 to 13	Formalized mathematics and computer science

Table 1: Course plan for the spring 2023 version of *Automated Reasoning* at DTU.

5.2 Research Questions

We evaluated the usefulness of PROOFBUDDY by attempting to answer the following research questions (RQ), which cover both perspectives of the evaluation:

RQ1 How many resources does PROOFBUDDY need to be usable by many students at once?

RQ2 What functionality do students miss in PROOFBUDDY compared to Isabelle?

RQ3 What issues did students encounter when interacting with PROOFBUDDY?

RQ4 Is the data collected by PROOFBUDDY useful for conducting didactic studies?
Sample research questions in didactic studies include:

- **SQ1** Which types of mistakes do students make while writing functional programs?
- **SQ2** Which types of mistakes do students make while writing proofs?
- **SQ3** Which types of feedback are immediately useful to students?
- **SQ4** How often do students use the possibility to get feedback from the type checker?
- **SQ5** How long do students need for an exercise?

Research question 4 includes a number of sample questions (SQ) which we imagine that researchers could want to answer when conducting a didactic study. It is important to note that obtaining answers to these questions would not by itself show anything about the efficacy of proof assistants in education as compared to pen-and-paper. Instead, answering these questions for multiple groups of students taught using different approaches (e.g. various approaches to using proof assistant or with pen-and-paper) could provide comparable measures which could potentially be used to show differences in efficacy. PROOFBUDDY enables such studies by allowing researchers to collect data about student interaction with proof assistants, but other approaches are still needed to collect data about student behavior using pen-and-paper, e.g. to conduct a randomized controlled trial to determine the efficacy of using proof assistants as compared to pen-and-paper.

5.3 Methods

We conducted the evaluation by asking the participants to solve a number of exercises using PROOFBUDDY and then fill out a short questionnaire. The exercises consisted of two parts: proving formulas directly in the natural deduction system of Isabelle/Pure, and programming and proving the correctness of simple program optimizations for an imperative programming language. The first and second authors instructed and supervised the students during the evaluation.

5.3.1 Population

The population studied in the evaluation consists of the students enrolled in the spring 2023 version of the course *Automated Reasoning*, of which there were 19. We included only those students who were physically present at the exercise session on March 17 in the evaluation. We recruited students by asking them to participate before the exercise session. Participants were not reimbursed financially or otherwise. Before starting the evaluation, we briefed participants about the purpose of PROOFBUDDY and the overall elements of the evaluation, including the research questions and the extent of data collected. Participants were not briefed about the interface of PROOFBUDDY such that we could study the discoverability of the features. Participants were not debriefed after the evaluation. 12 students opted to participate in the evaluation.

5.3.2 Ethical Considerations

We did not collect any personal identifiable information of participants during the evaluation, but did collect all text entered into PROOFBUDDY. We briefed participants about the extent of the data collection before the start of the evaluation. We considered students to have given informed consent to participate in the evaluation when they had listened to the briefing, read a letter describing their rights as participants, and started using PROOFBUDDY. Students in the course were free to choose not to take part in the evaluation and instead solve the exercises using their usual means instead of PROOFBUDDY. The exercises used for the evaluation were not used for grading students.

5.3.3 Threats to validity

Since the population consists of students who have voluntarily selected to follow the course *Automated Reasoning*, we may experience some selection bias. More importantly, the students following the course already had some experience with Isabelle, and might not experience the same issues as complete beginners. Our preliminary usability study is thus not generalizable to beginners. Additional selection bias may be introduced since there may be a correlation between students who actively follow the course (and so are present during the evaluation) and those who do not. The use of a questionnaire where we ask participants to report their own opinions on the importance of missing features and technical issues with PROOFBUDDY may introduce self-report bias. Manually categorizing participant questions asked during the evaluation may introduce researcher bias.

5.3.4 Analyses

To answer RQ1, we monitored the performance of the PROOFBUDDY tool and the resource usage (in terms of CPU and memory usage) of the server hosting the tool while performing the evaluation.

To answer RQ2 and RQ3 we asked participants to fill out a questionnaire about their experiences and any issues they may have encountered while using the PROOFBUDDY tool. We additionally recorded the questions participants asked the instructors while using PROOFBUDDY. The questionnaire contained the following questions (questions 1–3 concerning RQ2 and questions 4–5 concerning RQ3):

1. Do you usually use Isabelle/jEdit or Isabelle/VSCode?
2. Did you miss any features of your usual editor while using PROOFBUDDY?
3. If yes, how important do you think each of those features are on a scale from 1 to 5 (with 1 being not important and 5 being very important)?

4. Did you encounter any technical issues while using ProofBuddy?
5. If yes, how much did each of these issues affect your work on a scale from 1 to 5 (with 1 being not at all and 5 being very much)?

To answer RQ4, we used the data collected by ProofBuddy during the evaluation to set up mock analyses for the sample questions in Section 5.2. By setting up analyses for each of the sample questions we determined whether the data collected by ProofBuddy was adequate to perform analyses in didactic studies. The data collected by ProofBuddy includes error messages and warnings from the type checker and the proof checker of Isabelle. The syntactic, type level and tactic level mistakes students make while writing functional programs and proofs can be categorized using this data. The data also includes the actual programs and proofs, and can thus also be used to categorize semantic mistakes, i.e. programs that type check but do not have correct behavior or proofs that prove a different theorem than what was intended. SQ1 and SQ2 could be answered by categorizing the mistakes students make and ranking the categories by frequency. The data collected by ProofBuddy contains timestamps and includes both the actual theory and any messages from Isabelle. This data can be used to trace the evolution of the theory over time, noting the messages (e.g. errors and warnings) ProofBuddy has given the user between each step, and thus determining which messages lead to theories with fewer mistakes. SQ3 could be answered by categorizing the mistakes and messages, then ordering them chronologically and measuring the association between various messages and the disappearance of mistakes in the proofs. This is of course a simplified view of the causality between messages and mistakes, and studying the actual proof scripts in more detail could enable more refined analyses. The data collected by ProofBuddy consists of records of each instance of a student asking the tool for feedback. SQ4 could thus be answered by counting how many records exist in the dataset for each student. If we assume that students progress immediately from one exercise to the next, SQ5 could be answered by estimating the time spent on each exercise.

5.4 Observation Protocol

Nearly all students started the exercise session using ProofBuddy. One student had problems logging in, but trying a new account and password resolved the issue. At first all students got familiar with the tool and asked questions about the interface. Most of the students only dealt with the logic exercises, but some started with the program optimization exercise. During the exercise session we collected the questions the students asked the instructors because there was nearly no interaction between the students. We sorted the observed questions into the following categories:

1. **Creating proofs:** Problems during the process of proving, including recognizing and understanding assumptions and subgoals as well as how to start a proof. Questions about whether a proof is complete and correct also belong to this category;
2. **Mathematical inscriptions:** Problems with mathematical inscriptions, like how to write a proof step or argument in formal language;
3. **ProofBuddy usability:** Problems with the web interface of ProofBuddy, like menu functions and understanding the meaning of displayed elements;
4. **Working with Isabelle:** Problems with Isabelle, like choosing the correct rule, variable or structure and keywords of the Isar language, including typing errors;
5. **Logic:** Problems with aspects of logic, like syntax and semantics of definitions or the usage of quantifiers and other connectives;

6. **Functions:** Problems with aspects of functional programming, like syntax and semantics of functions or the concept of functions.

Next, we summarize the kinds of questions observed in each category:

1. **Creating proofs:** There was only one question related to induction.
2. **Mathematical inscriptions:** The students had no questions in this category.
3. **ProofBuddy Usability:** There were 17 question in this category, concerning: whether there is a difference in the syntax of Isabelle and ProofBuddy, how to use the special symbol keyboard, how to load and check theories (five questions).
4. **Working with Isabelle:** There were 19 questions in this category, concerning: syntax of Isabelle, the usage of strings in Isabelle and the structure of induction in Isabelle.
5. **Logic:** There were 21 questions in this category, concerning: scope of quantifiers, the freshness of variables when eliminating an existential quantifier (using the obtain keyword) and contradiction.
6. **Functions:** There were two questions in this category, concerning: conceptional understanding of functions, their semantics and programming.

In the first hour, most questions concerned the usage of ProofBuddy and the checking of theories. We observed that ProofBuddy needed a long time to answer the checking requests and therefore the students had to wait for feedback. For that reason students started using their own installations of Isabelle to check the proofs, but kept writing their proofs using ProofBuddy. The students seemed to like interacting with ProofBuddy, and several noted especially the possibility to choose a rule from the drop-down menu, without searching for the right rule among many irrelevant rules.

After the first half hour there were many questions about the syntax of Isabelle and logic. After one and a half hours the students worked on their own and asked only a few further questions. The first students left after two hours, and 6 students were still working after two and a half hours when the evaluation ended. All participants except one filled out the questionnaire before leaving.

5.5 Data Analysis

We analyzed the logged communication between frontend, backend and Isabelle Server (see Figure 5). This reveals that the average processing time of a request in the backend takes 16 seconds, where the handling of the request by the web server, i.e. writing the development to a theory file and attaching dependencies to the Isabelle request, takes only one second. The Isabelle Server used the remaining 15 seconds to check the development and compute the feedback. We did not notice any spikes in the processing time. Furthermore we analyzed the memory and CPU utilization of the server. Figure 4 shows the percentage of memory used during the evaluation. There is an increase in memory usage when starting ProofBuddy and another one when the students begin to log in. The measurement of the CPU utilization always resulted in the same value (0.64% at the user level).

The questionnaire included two different validated standard questionnaires and questions comparing Isabelle and ProofBuddy. 11 of 12 participants completed the survey and the answers support the impression we got by observing the exercise session.

We used the validated short version questionnaire System Usability Scale (SUS) [6] where ProofBuddy obtains an average score of 65/100. The students find ProofBuddy easy to use but do not see themselves using the tool frequently.

The second questionnaire used is the short version of the User Experience Questionnaire (UEQ) [32]. In this questionnaire, values between -0.8 and 0.8 represent a neutral evaluation of the corresponding

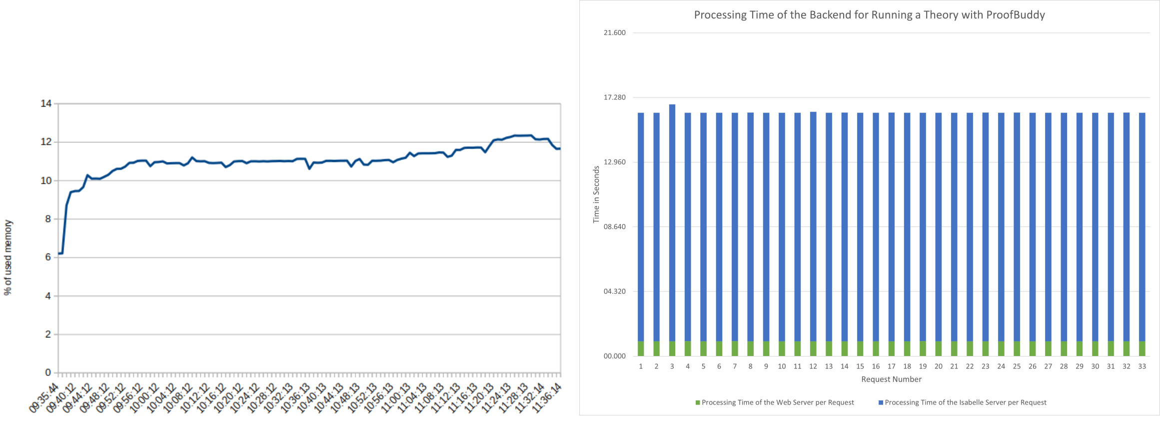


Figure 4: Utilization of the server memory when running ProofBuddy.

Figure 5: Vertical diagram displaying the processing time of the backend for checking theories with ProofBuddy.

scale, values > 0.8 represent a positive evaluation and values < -0.8 represent a negative evaluation. The range of the scales is between -2 (horribly bad) and +2 (extremely good). The pragmatic quality has a value of 0.775 where the efficiency has a bad score and the hedonic quality has a value of 0.727.

The rest of the questionnaire concerned the comparison of ProofBuddy and Isabelle. Most of the students normally use Isabelle/jEdit and not Isabelle/VSCode. In comparison with Isabelle, the students miss the actual proof state and the immediate feedback and criticize the long waiting time and the error detection. There was a problem switching between the different activities, which overrides the editor content without saving the previous content anywhere. Sometimes the cursor jumps to the beginning of the editor. Often the students do not find the symbol keyboard and sometimes the keyboard overlaps with the editor or output panes. It sometimes happens that ProofBuddy does not detect all errors in a theory, even when copying the theory into Isabelle resulted in errors. Some students used ProofBuddy to write proofs but checked them in Isabelle/jEdit. Some students mentioned that ProofBuddy never gave them any feedback and instead seemed to become stuck when asked to check a theory.

5.6 Discussion

As we observed, students like the web interface of ProofBuddy, especially the dropdown menu. They find it intuitive to use the interface but there are some issues.

The main issue observed with the usability of ProofBuddy was the slow checking of theories. The bad score in efficiency in the UEQ follows from the long waiting time which was the main usability problem. Therefore students state in the SUS that they will not use ProofBuddy frequently. Additionally, some students reported that ProofBuddy did not report errors in their theories, indicating another issue with the communication between ProofBuddy and Isabelle. We note also that the measurement of CPU utilization always having the same value might have some connection to this issue.

Since the evaluation, the issues concerning the response time have been fixed. Firstly, there was an error in processing and combining chunks of large responses from the Isabelle server, in which case ProofBuddy appeared to check the theory indefinitely. Secondly, we specified the headless session option for the delay to consolidate the status of command evaluation (“headless_consolidate_delay”). The “headless_consolidate_delay”, which is by default set to 15 seconds, seems to correlate to the response

time of 15 seconds by the Isabelle server. By reducing the "headless_consolidate_delay" to 0.5 seconds, we observed a much shorter response time from the Isabelle server and therefore from PROOFBUDDY. During the evaluation, users were only able to access recently saved theories through the file browser. Reloading the page thus resulted in losing any progress made, since the users were unable to access the saved theories on the server. Users are now able to access saved theories through the file browser.

Despite the usability issues of PROOFBUDDY, the data collection features worked as expected. All of the data required to answer the sample question as described above was thus present in the PROOFBUDDY database. Unfortunately, the usability issues meant that many students stopped using PROOFBUDDY and returned to using their own Isabelle instead. In a real study this would of course not be acceptable, and so our first priority is remedying the observed usability issues.

6 Future Work

We plan to improve PROOFBUDDY. One approach for optimizing the response time is pre-assessing the Isabelle theories in the frontend. The frontend is already equipped with a parser for the Isabelle language. This opens the possibility to assess the theories for syntactical errors, such as missing brackets, or failures in the Isar proof syntax where special keywords are missing, prior to sending them to the backend. Only sending syntactically correct theories would reduce the server workload, which could improve the scalability of the PROOFBUDDY backend.

We additionally plan to extend the course management functionality by adapting the linter, the drop-down menu and the symbol keyboard to more exercises to support the declared learning objectives of the associated learning activities. Furthermore, we need to improve the file management, such that there is a database which stores courses and teacher profiles and for each student their own files within their profile. PROOFBUDDY already saves the Isabelle theory versions on the server explicitly, if the user presses the upload button, or implicitly, prior to the checking by Isabelle.

Another step will be to analyze the collected data automatically in order to answer the research questions: (1) what kind of feedback do students need during the proof writing process and (2) which parts of the proof are causing the students difficulties. By answering the first research question, the feedback of Isabelle can be extended to give more precise feedback depending on the kind of mistake. The problems discovered by answering the second research question could be solved by a hint to solve an exercise (adaptive learning) or by explaining a concept in the lecture one more time.

We have developed a new BSc-level course at TUB, where we focus on teaching how to create and properly write down proofs. The course is planned for summer 2023. Referring to our research questions above, the concept of the course is to enable substantially more feedback for students. Thus, we use PROOFBUDDY in addition to the help of the instructor and fellow students to provide the feedback much more directly (in real time) and also more individually. The idea is to start with propositional logic and first-order logic, because knowledge about these is a good foundation for structuring proofs [33]. Afterwards, we introduce inductive data structures and prove properties about them. We want to avoid that students just learn to use the tool without actually understanding the proofs that they develop with it. Therefore, the concept of the course aims to strengthen the mutual transformation between formal proofs—as developed with PROOFBUDDY—and traditional pen-and-paper proofs, in both directions. Confronted with these transformations, students are supposed to also learn that there are different degrees of formality to prove propositions. This course offers the opportunity to evolve the kind of feedback a proof assistant should give to learners that it assists during the learning process. We plan to iteratively change the feedback during several semesters and analyze if the students have fewer problems with the

exercises. Furthermore we will test new features in a Bachelor course dealing with graph structure theory at TUB in summer 2023, specifically for exercises about tree width. The students in this course have not used a proof assistant before. The focus of this study lies on the time students need to step into a proof assistant and how exercises have to be prepared such that students can manage the formal language.

There will also be an introductory course in theorem proving with Isabelle at TUB in summer 2023 where we start with PROOFBUDDY to introduce propositional logic, first-order logic and sets. The exercises will be solved in Isabelle/jEdit with the opportunity to use PROOFBUDDY instead.

We would also like to carry out a study about the impact of a proof assistant for learning proofs. This could be measured in an experiment where one group is taught with PROOFBUDDY and a control group only learns the topic with pen-and-paper proofs. But for this kind of study one has to measure the learning of proof competences in a fair way, which can be difficult as detailed above.

7 Conclusion

We introduce PROOFBUDDY, a web-based tool for guiding and monitoring the use of Isabelle/HOL by students. Proof assistants like Isabelle allow students to write proofs about functional programs, and many concepts in, e.g. logic can naturally be encoded as functional programs.

Unfortunately, not much evidence has yet been collected about the efficacy of using proof assistants in education. We have identified that one of the main issues in doing so seems to be a lack of tools for studying the interactions of students with proof assistants. Additionally, instructors have reasonable worries about teaching with very expressive languages, which may be difficult to learn and understand. PROOFBUDDY provides the opportunity to restrict the expressivity of Isabelle depending on the exercise and can therefore be used as a learning and teaching tool. Furthermore PROOFBUDDY allows researchers to collect data about student interactions for conducting studies. We log the data of the communication between frontend, backend and the Isabelle server. The collected data allows us to improve the error messages of Isabelle and analyze at which parts in a proof the students fail. Hence, we can offer hints and additional feedback. Such an approach allows instructors to gain insight into how students try to write and prove properties about inductive definitions and use pattern matching; it also enables us to carry out didactic research on student behavior. We expect that instructors could use the results of such studies to design guided exercises which support the learning progression of individual students.

References

[1] Vincent A.W.M.M. Aleven & Kenneth R. Koedinger (2002): *An effective metacognitive strategy: learning by doing and explaining with a computer-based Cognitive Tutor*. *Cognitive Science* 26(2), pp. 147–179, doi:10.1207/s15516709cog2602_1.

[2] Jeremy Avigad (2019): *Learning Logic and Proof with an Interactive Theorem Prover*. In Gila Hanna, David A. Reid & Michael de Villiers, editors: *Proof Technology in Mathematics Research and Teaching*, *Mathematics Education in the Digital Era* 14, Springer International Publishing, Cham, pp. 277–290, doi:10.1007/978-3-030-28483-1_13.

[3] Yves Bertot & Pierre Castéran (2004): *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* Springer Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.

[4] Sebastian Böhne, Maria Knobelsdorf & Christoph Kreitz (2016): *Mathematisches Argumentieren und Beweisen mit dem Theorembeweiser Coq*. In A. Schwill & U. Liuckey, editors: *HDI 2016 – 7. Fachtagung zur Hochschuldidaktik der Informatik*, *Commentarii informaticae didacticae* 10, Universitätsverlag Potsdam, pp. 69–80. Available at `http://eprints.cs.univie.ac.at/6838/`. (in German).

[5] Sebastian Böhne & Christoph Kreitz (2018): *Learning how to Prove: From the Coq Proof Assistant to Textbook Style*. In Pedro Quaresma & Walther Neuper, editors: *Theorem proving components for Educational software*, *Electronic Proceedings in Theoretical Computer Science* 267, Open Publishing Association, pp. 1–18, doi:10.4204/eptcs.267.1.

[6] John Brooke (1996): *SUS: A quick and dirty usability scale*. In Patrick W. Jordan, B. Thomas, Ian Lyall McClelland & Bernard Weerdmeester, editors: *Usability Evaluation in Industry*, chapter 21, CRC Press, London, England, pp. 189–195, doi:10.1201/9781498710411.

[7] Esther Brunner (2014): *Mathematisches Argumentieren, Begründen und Beweisen*. Springer Spektrum Berlin, Heidelberg, doi:10.1007/978-3-642-41864-8. (in German).

[8] Michael B. Burke (2006): *Electronic Media Review: Logic and Proofs (Web-based course)*. *Teaching Philosophy* 29(3), pp. 255–260, doi:10.5840/teachphil200629327.

[9] Kevin Buzzard & Mohammed Pedramfar (2021): *The Natural Number Game*. Available at `https://www.ma.imperial.ac.uk/%7Ebuzzard/xena/natural_number_game/`.

[10] Nathan C. Carter & Kenneth G. Monks (2013): *Lurch: a word processor built on OpenMath that can check mathematical reasoning*. In Christoph Lange, David Aspinall, Jacques Carette, James Davenport, Andrea Kohlhase, Michael Kohlhase, Paul Libbrecht, Pedro Quaresma, Florian Rabe, Petr Sojka, Iain Whiteside & Wolfgang Windsteiger, editors: *MathUI, OpenMath, PLMMS and ThEdu Workshops and Work in Progress at the Conference on Intelligent Computer Mathematics*, *CEUR Workshop Proceedings* 1010, Aachen, pp. 23:1–23:10. Available at `http://ceur-ws.org/Vol-1010/paper-23.pdf`.

[11] Nathan C. Carter & Kenneth G. Monks (2013): *Lurch: a word processor that can grade students' proofs*. In Christoph Lange, David Aspinall, Jacques Carette, James Davenport, Andrea Kohlhase, Michael Kohlhase, Paul Libbrecht, Pedro Quaresma, Florian Rabe, Petr Sojka, Iain Whiteside & Wolfgang Windsteiger, editors: *MathUI, OpenMath, PLMMS and ThEdu Workshops and Work in Progress at the Conference on Intelligent Computer Mathematics*, *CEUR Workshop Proceedings* 1010, Aachen, pp. 4:1–4:10. Available at `http://ceur-ws.org/Vol-1010/paper-04.pdf`.

[12] Nathan C. Carter & Kenneth G. Monks (2017): *A Web-Based Toolkit for Mathematical Word Processing Applications with Semantics*. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe & Olaf Teschke, editors: *Intelligent Computer Mathematics*, *Lecture Notes in Computer Science* 10383, Springer International Publishing, Cham, pp. 272–291, doi:10.1007/978-3-319-62075-6_19.

[13] Ian Fette & Alexey Melnikov (2011): *The WebSocket Protocol*. Technical Report 6455, Internet Engineering Task Force, doi:10.17487/RFC6455.

[14] Christiane Frede & Maria Knobelsdorf (2018): *Explorative Datenanalyse der Studierendenperformance in der Theoretischen Informatik*. In N. Bergner, R. Röpke, U. Schroeder & D. Krömker, editors: *Hochschuldidaktik der Informatik - HDI 2018 - 8. Fachtagung des GI-Fachbereichs und Ausbilding/Didaktik der Informatik, Frankfurt, Germany, September 12-13, 2018*, Universitätsverlag Potsdam, pp. 135 – 149. Available at `http://eprints.cs.univie.ac.at/6870/`. (in German).

[15] Asta Halkjær From, Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *SeCaV: A Sequent Calculus Verifier in Isabelle/HOL*. In Mauricio Ayala-Rincon & Eduardo Bonelli, editors: Proceedings 16th *Logical and Semantic Frameworks with Applications*, Buenos Aires, Argentina (Online), 23rd - 24th July, 2021, *Electronic Proceedings in Theoretical Computer Science* 357, Open Publishing Association, pp. 38–55, doi:10.4204/EPTCS.357.4.

[16] Aiso Heinze & Kristina Reiss (2003): *Reasoning and Proof: Methodological Knowledge as a Component of Proof Competence*. In Maria Alessandra Mariotti, editor: *Proceedings of the Third Conference of the European Society for Research in Mathematics Education*, pp. 1–10. Available at `http://www.lettredelapreuve.org/OldPreuve/CERME3Papers/Heinze-paper1.pdf`. Thematic Working Group 4, paper 5.

[17] Maxim Hendriks, Cezary Kaliszyk, Femke van Raamsdonk & Freek Wiedijk (2010): *Teaching logic using a state-of-the-art proof assistant*. *Acta Didactica Napocensia* 3(2), pp. 35–48. Available at `http://dppd.ubbcluj.ro/adn/article_3_2_4.pdf`.

[18] Frederik Krogsdal Jacobsen & Jørgen Villadsen (2022): *Teaching Functional Programmers Logic and Metatheory*. In Peter Achten & Elena Machkasova, editors: *Trends in Functional Programming In Education*, *Electronic Proceedings in Theoretical Computer Science* 363, Open Publishing Association, pp. 74–92, doi:10.4204/eptcs.363.5.

[19] Felix Kiehn, Christiane Frede & Maria Knobelsdorf (2017): *Was macht Theoretische Informatik so schwierig? Ergebnisse einer qualitativen Einzelfallstudie*. In Maximilian Eibl & Martin Gaedke, editors: *INFORMATIK 2017*, Gesellschaft für Informatik, Bonn, pp. 267–278, doi:10.18420/in2017_20. (in German).

[20] Maria Knobelsdorf & Christiane Frede (2016): *Analyzing Student Practices in Theory of Computation in Light of Distributed Cognition Theory*. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, ACM, New York, NY, USA, pp. 73–81, doi:10.1145/2960310.2960331.

[21] Maria Knobelsdorf, Christiane Frede, Sebastian Böhne & Christoph Kreitz (2017): *Theorem Provers as a Learning Tool in Theory of Computation*. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, Association for Computing Machinery, New York, NY, USA, pp. 83–92, doi:10.1145/3105726.3106184.

[22] Graham Leach-Krouse (2017): *Carnap: An Open Framework for Formal Reasoning in the Browser*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software*, *Electronic Proceedings in Theoretical Computer Science* 267, Open Publishing Association, pp. 70–88, doi:10.4204/EPTCS.267.5.

[23] Christoph Lüth & Martin Ring (2013): *A Web Interface for Isabelle: The Next Generation*. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka & Wolfgang Windsteiger, editors: *Intelligent Computer Mathematics*, *Lecture Notes in Artificial Intelligence* 7961, Springer, pp. 326–329, doi:10.1007/978-3-642-39320-4_22.

[24] Maria Alessandra Mariotti (2019): *The Contribution of Information and Communication Technology to the Teaching of Proof*. In Gila Hanna, David A. Reid & Michael de Villiers, editors: *Proof Technology in Mathematics Research and Teaching*, *Mathematics Education in the Digital Era* 14, Springer International Publishing, Cham, pp. 173–195, doi:10.1007/978-3-030-28483-1_8.

[25] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Doorn & Jakob Raumer (2015): *The Lean Theorem Prover (System Description)*. In Amy P. Felty & Aart Middeldorp, editors: *Automated Deduction - CADE-25*, *Lecture Notes in Computer Science* 9195, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[26] Tobias Nipkow (2012): *Teaching Semantics with a Proof Assistant: No more LSD Trip Proofs*. In V. Kuncak & A. Rybalchenko, editors: *Verification, Model Checking, and Abstract Interpretation*, *Lecture Notes in Computer Science* 7148, Springer, pp. 24–38, doi:10.1007/978-3-642-27940-9_3.

[27] Tobias Nipkow (2022): *Programming and Proving in Isabelle/HOL*. Available at `https://isabelle.in.tum.de/doc/prog-prove.pdf`.

[28] 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.

[29] Siddhartha Prasad, Ben Greenman, Tim Nelson, John Wrenn & Shriram Krishnamurthi (2022): *Making Hay from Wheats: A Classsourcing Method to Identify Misconceptions*. In Ilkka Jormanainen & Andrew Petersen, editors: *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*, Koli Calling '22, Association for Computing Machinery, New York, NY, USA, pp. 2:1–2:7, doi:10.1145/3564721.3564726.

[30] Martin Ring & Christoph Lüth (2014): *Real-time collaborative Scala development with Clide*. In Heather Miller & Philipp Haller, editors: *Proceedings of the Fifth Annual Scala Workshop*, ACM, pp. 63–66, doi:10.1145/2637647.2637652.

[31] Anders Schlichtkrull, Jørgen Villadsen & Asta Halkjær From (2018): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software*, *Electronic Proceedings in Theoretical Computer Science* 290, pp. 1–13, doi:10.4204/EPTCS.290.1.

[32] Martin Schrepp, Andreas Hinderks & Jörg Thomaschewski (2017): *Design and Evaluation of a Short Version of the User Experience Questionnaire (UEQ-S)*. *International Journal of Interactive Multimedia and Artificial Intelligence* 4(6), pp. 103–108, doi:10.9781/ijimai.2017.09.001.

[33] John Selden & Annie Selden (2009): *Teaching Proving by Coordinating Aspects of Proofs with Students' Abilities*. In Despina A. Stylianou, Maria L. Blanton & Eric J. Knuth, editors: *Teaching and Learning Proof Across the Grades*, chapter 20, Routledge, pp. 339–354, doi:10.4324/9780203882009.

[34] Wilfried Sieg (2007): *The AProS Project: Strategic Thinking & Computational Logic*. *Logic Journal of the IGPL* 15(4), pp. 359–368, doi:10.1093/jigpal/jzm026.

[35] Socket.IO (2023): `https://socket.io/`. Online. Accessed April 18 2023.

[36] Alexander Steen, Max Wisniewski & Christoph Benzmüller (2016): *Einsatz von Theorembeweisern in der Lehre*. *Commentarii informaticae didacticae* 10, pp. 81–92. Available at `https://orbilu.uni.lu/bitstream/10993/40839/1/cid10.pdf`. (in German).

[37] Bernard Sufrin & Richard Bornat (1996): *User interfaces for generic proof assistants part I: Interpreting gestures*. Available at `https://www.cs.ox.ac.uk/people/bernard.sufrin/jape.org.uk/DOCUMENTS/CURRENT/UITP96-1.pdf`.

[38] Bernard Sufrin & Richard Bornat (1998): *User interfaces for generic proof assistants part II: Displaying proofs*. Available at `https://www.cs.ox.ac.uk/people/bernard.sufrin/jape.org.uk/DOCUMENTS/CURRENT/UITP96-2.pdf`.

[39] Athina Thoma & Paola Iannone (2022): *Learning about proof with the theorem prover LEAN: The abundant numbers task*. *International Journal of Research in Undergraduate Mathematics Education* 8, pp. 64–93, doi:10.1007/s40753-021-00140-1.

[40] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction and the Isabelle Proof Assistant*. In Pedro Quaresma & Walther Neuper, editors: *Theorem proving components for Educational software*, *Electronic Proceedings in Theoretical Computer Science* 267, Open Publishing Association, pp. 140–155, doi:10.4204/eptcs.267.9.

[41] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2019): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Theorem proving components for Educational software*, *Electronic Proceedings in Theoretical Computer Science* 290, Open Publishing Association, pp. 14–29, doi:10.4204/eptcs.290.2.

[42] Jørgen Villadsen, Alexander Birch Jensen & Anders Schlichtkrull (2015): *NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle*. *Journal of Applied Logics* 4(1), pp. 55–82. Available at `http://www.collegepublications.co.uk/downloads/ifcolog00010.pdf`.

[43] Jelle Wemmenhove, Thijs Beurskens, Sean McCarren, Jan Moraal, David Tuin & Jim Portegies (2022): *Waterproof: educational software for learning how to write mathematical proofs*. arXiv:2211.13513.

[44] Markus Wenzel (2022): *The Isabelle system manual*. Available at `https://isabelle.in.tum.de/doc/system.pdf`.

[45] Jack Wrenn & Shriram Krishnamurthi (2021): *Reading Between the Lines: Student Help-Seeking for (Un)Specified Behaviors*. In Otto Seppälä & Andrew Petersen, editors: *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*, Koli Calling '21, Association for Computing Machinery, New York, NY, USA, pp. 14:1–14:6, doi:10.1145/3488042.3488072.

[46] John Wrenn & Shriram Krishnamurthi (2019): *Executable Examples for Programming Problem Comprehension*. In Robert McCartney, Andrew Petersen, Anthony Robins & Adon Moskal, editors: *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, Association for Computing Machinery, New York, NY, USA, pp. 131–139, doi:10.1145/3291279.3339416.

[47] John Wrenn & Shriram Krishnamurthi (2020): *Will Students Write Tests Early Without Coercion?* In Nick Falkner & Otto Seppälä, editors: *Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, Koli Calling '20, Association for Computing Machinery, New York, NY, USA, pp. 27:1–27:5, doi:10.1145/3428029.3428060.

Computer Aided Design and Grading for an Electronic Functional Programming Exam

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 nor grading fairness in favor of automation. To advance automation in the design and fair grading of (functional programming) e-exams, we introduce the following: A novel algorithm to check *Proof Puzzles* based on finding correct sequences of proof lines that improves fairness compared to an existing, edit distance-based algorithm; an open-source static analysis tool to check source code for task relevant features by traversing the abstract syntax tree; a higher-level language and open-source tool to specify regular expressions that makes creating complex regular expressions less error-prone. Our findings are embedded in a complete experience report on transforming a paper exam to an e-exam. 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. Almost all tasks can be graded automatically at least in part (correct solutions can almost always be detected as such), the students agree that an e-exam is a fitting examination format for the course but are split on how well they can express their thoughts compared to a paper exam, and examiners enjoy a more time-efficient grading process while the point distribution in the exam results was almost exactly the same compared to a paper exam.

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 [13] nor constructive alignment (CA) [3] in favor of automation. For example, an exam consisting only of multiple choice tasks would be easy to grade automatically, but it would also be impossible to assess the ability of students to *create* a solution rather than *select* a given one. On the other hand, it is possible to integrate arbitrary program logic into an e-exam, which does not only help with automating grading (checking the correct answer through means ranging from simple equality checks, over specialized, self-written algorithms, to possibly even artificial intelligence [26]), but can also be used for task design (e.g., randomization to impede cheating, sensibly showing or hiding input fields depending on previously-given answers, execution of student code to allow students to test their answers in a controlled way). Following the recent introduction of large scale electronic examinations [28] and the

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

Elena Machkasova (Ed.): International Workshop on Trends in Functional Programming in Education 2023 (TFPIE 2023)
EPTCS 382, 2023, pp. 22–44, doi:10.4204/EPTCS.382.2

© O. Lübke, K. Fuger, F. H. Bahnsen, K. Billerbeck, S. Schupp
This work is licensed under the Creative Commons Attribution-Share Alike License.

development of the extensible *Your Open Examination System for Activating and emPowering Students* (YAPS) [1] at Hamburg University of Technology (TUHH), we investigate in this paper how to leverage that opportunity to improve the examination in our functional programming (FP) course. We provide a complete tour of how we successfully designed an e-exam for the FP course at TUHH, both from a technological and an educational perspective.

As a first step we analyzed our pre-e-exam FP courses and past exams to identify potentials for automation and improvement. The analysis was driven by the learning objectives (LOs) of the course and how past exam tasks aligned with them. We found that regular expressions (RegExs) are a good candidate for automatically checking answers that consist of short source code snippets. More elaborate programming tasks could be evaluated using software tests. One particular LO (Students are able to interpret compiler warnings and errors) had low coverage, but in the e-exam we could alleviate that by integrating live compiler feedback. Overall, we found that many of our existing tasks could reasonably be transformed to an e-exam version.

We implemented our ideas in YAPS, extending the system where necessary. Because we teach FP using the example of Haskell but YAPS only offered a C/C++ compiler, we extended it with a common template for Haskell programming tasks. The template facilitates the quick creation of new tasks, and 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 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 Puzzles*) that are then graded by an algorithm based on edit distance between correct solution and given answer. This 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 in line 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, e.g., for taking notes, or noting assumptions they made in case they deem the task ambiguous.

For evaluation, we manually analyzed the degree of automation of the e-exam (focusing on automated grading), and asked the students that participated in the exam for their opinion. Additionally, we report our own experiences and reflect on sources of errors in automated grading. Grading of almost all tasks features at least some automated elements; some tasks are graded fully automated. For almost all tasks correct answers can be awarded the full amount of points automatically. Yet, awarding a sensible amount of points for partially correct solutions remains a challenge in most cases. For us, the most common mistakes in automated grading were related to tasks checked by RegExs and programming tasks. Currently, experience and manual testing are the best defense against errors, but our RegEx creation tool is a first step towards more (but less error-prone) automation in that regard as well. The results from the student poll are encouraging, as most students state that an e-exam is the right format for the FP course. Yet, they are split on how well they can express their thoughts in comparison to a paper exam.

Altogether, we contribute the following to advancing computer aided design and grading for FP e-exams:

- An algorithm to check *Proof Puzzles* (cf. Section 5.3).
- A small static analysis tool to check Haskell code for exam task relevant features (cf. Section 5.5).

- A language to specify RegExs on Haskell snippets (cf. Section 5.6).

In addition, this paper is a complete experience report which highlights challenges when moving from a pen-and-paper to an e-exam, and provides answers to the following issues:

- *How can exam quality be ensured?* By analysis of the LOs of each task. The e-exam should assess the same LOs as the paper exam (cf. Section 4).
- *How can fair grading be ensured?* By critically examining results from automated grading (cf. Section 5.3), giving students room to express their ideas (cf. Section 5.4), and (for now) resorting to manual assessment if necessary (cf. Sections 4.3 and 6.1).
- *How can programming tasks with compiler support be realized in a way that feels natural to students, does not leak secret information about the task, and facilitates automation?* By employing a reusable template where the `main` function cannot be edited by students (cf. Section 5.1).
- *How can certain restrictions in programming tasks be checked?* By syntactic analysis of the submitted code (cf. Section 5.5).
- *How can an empty answer be distinguished from the absence of an answer?* By combining ambiguous input fields with a suitable yes/no question (cf. Section 5.2).
- *How can the absence of errors be ensured in complex RegExs that are used to check answers?* By employing a higher-level, domain-specific language that compiles to RegExs (cf. Section 5.6).

The paper is organized as follows: Section 2 discusses related work, and Section 3 introduces constructive alignment (CA) and the YAPS examination software. Sections 4 to 6 contain the above-mentioned analysis, realization, and evaluation respectively. Section 7 summarizes and provides an outlook on future work.

2 Related Work

While electronic assessment and automated grading have been an active research area for decades [8, 21], the design of entire electronic exams (e-exams) seems to get less attention. The report on introducing e-exams in two (Java-based) programming courses by Rajala et al. [25] appears closest to our work. They also establish a categorization of tasks, focus on complete exam designs, and use questions similar to ours to collect student feedback. The most notable difference is the method to ensure exam quality: the authors asked two (otherwise uninvolved) other researchers for their opinion. Instead, we suggest to make sure all learning objectives (LOs) of the course are assessed in the exam through analysing each task. For additional quality assurance, both methods could be combined. Other differences include that Rajala et al. require source code submitted in the exam to compile (which we decided against), check correctness by comparing the program output (instead, we use property tests), and they neither restrict which code can be executed, nor the input language. These differences can likely be attributed to different LOs in the respective courses and the different programming languages and paradigms.

Bloom's Taxonomy [13] is a tool widely used to assess task complexity, and can therefore also be used to assess an entire exam. Still, Sheard et al. developed a scheme for classifying exam tasks using seven different features, targeted specifically at introductory computer science courses [27]. Our analysis is different in that its aim is not to characterize a single exam, but to make sure exam quality did not degrade from one to the next exam. It uses the LOs covered by each task as the only feature to make sure all tasks are still constructively aligned [3].

An older study focussing on the transition from paper to e-exam has been conducted by Stergiopoulos et al. for a course in *Electronic Physics* [29]. However, the set of investigated task types is limited to yes/no questions, multiple choice, and calculations (with numeric input that must lie in a specific range).

Regarding challenges when introducing e-exams, Kuikka et al. surveyed educators at Turku University of Applied Sciences for which requirements for introducing e-exams they see [15]. In contrast, we focus on challenges when those requirements are met and an actual e-exam is created.

During the COVID-19 pandemic, educators worldwide had to move their classes online and shared their experiences. Often, e-exams are also part of these reports, but with varying level of detail regarding task design and automated grading. Loftsson and Matthíasdóttir report on how they combined different online teaching tools to transform a first semester (Python-based) programming course [16]. They evaluate the changes in depth based on student surveys and exam results, but, except for the employed tools, no details on task design and automated grading were given. Kappelmann et al. focus on how students can be engaged in online teaching in an introductory (Haskell-based) functional programming (FP) course, and also provide details on task designs and automated grading [12]. For checking programming tasks, they also use property testing, and add unit tests as well as a novel `IO` mocking library. The latter can be seen as another way of controlling the execution of student code (we prevent `main` from being changed), yet there is no mention of restricting the input language for certain tasks. For checking proof tasks, they include an actual proof checker instead of *Proof Puzzles*.

The *Proof Puzzles* in *Your Open Examination System for Activating and emPowering Students* (YAPS) are a similar to *Proof Blocks* [23, 24]: students are presented with building blocks containing lines of the proof as well as distractors, and are supposed to construct the proof via drag-and-drop. In *Proof Blocks*, the proof structure is encoded as a directed acyclic graph (DAG), and the points to award for a given solution are calculated using an edit distance measure. *Proof Puzzles* also rely on edit distance for awarding partial points. We found that this algorithm may produce unfair results and consequently developed a new one. One the one hand, our algorithm is based on pre-defined correct block sequences which can be seen as non-branching DAGs. On the other hand, we use a different mechanism for awarding partial points. Generally, points are awarded for each correct block in a sequence. Apart from the sequences, entry points are specified to allow for reentering the correct proof, which allows for a fair distribution of points even when some mistakes were made. An alternative to such block-based automatic proof grading is using a theorem prover [12, 10]. For instance, Kappelmann et al. employed *Check Your Proof* (CYP) [1] in an FP exam [12]. The input language of CYP is intentionally close to Haskell, and it supports checking equational reasoning, proofs by structural induction, proofs by extensionality, case analysis, and computation induction. For the exam, they did not require the students to strictly adhere to the syntax of CYP. Still, similar to many tasks in our e-exam, correct solutions, if given in the correct syntax, can automatically be verified as such. Additionally, establishing a common syntax for proofs in the course also facilitated the manual grading of partially correct submissions. Another example of a theorem prover used in an exam is given by Jacobsen et al., who integrated Isabelle/HOL [20] in a course on automated reasoning and report similar experiences with respect to grading. In comparison to theorem provers, *Proof Puzzles* or *Blocks* allow for fully automated proof grading, but they cannot reason about the proof itself. Instead, they rely on pre-defined structural information, such as the DAG representation. Yet, the two options do not necessarily exclude each other, as shown by McCartin-Lim et al. [17]. They developed a graphical, graph-based user interface where proof assumptions and assertions need to be selected by students and dragged to the graph pane, where they can be connected with directed edges. In the background, a theorem prover checks for each each assertion whether its proof is complete,

[1] `https://github.com/noschinl/cyp`

and the outcome is indicated visually.

Automatically grading programming tasks is a vast field that has recently been reviewed by Paiva et al. [21]. While it is often easy to determine whether a given program is correct through traditional testing, it is much harder to assign a reduced amount of points to partially correct solutions. Our approach is to execute multiple tests and use a set of rules that map test results to points. However, manual inspection is still required for incorrect programs for two reasons: Programs that do not compile cannot be evaluated this way (but we also want to grade those), and there are cases where the assigned amount of points is too low because the tests did not cover a detail that we do want to award points for. Static analysis is a more formal way to measure the difference between a given and correct solution based on a graph representation derived from the abstract syntax tree (AST) and its usefulness has been shown in numerous studies [21]. We also use static analysis in the automated grading process, but not for assigning reduced points. Instead, it is employed to check whether a given solution can be a viable solution at all. Some of our programming tasks restrict the input language by forbidding/requiring the use of certain language features or functions. Since, to our knowledge, no tool exists that reports on the functions and language features used in a Haskell program, we devised our own.

For grading smaller tasks that do not require writing a full program or function, we use regular expressions (RegExs). RegExs are a known tool for automated assessment, with applications such as parsing output of student programs [19, 22], extracting information from student code [19, 30], or even grading free-form text answers [11]. In this work, we are mostly concerned with generating correct RegExs that represent valid solutions to a given task because writing complex RegExs that accept different answers is tedious and error-prone. Existing work on ensuring the correctness of RegExs has recently been reviewed by Zheng et al. [31]. While there are numerous approaches to testing or even verifying existing RegExs, methods for creating RegExs are mostly concerned with learning them either from examples or from a natural language description. Another line of research is concerned with making RegExs easier to understand for humans, e.g., by abstraction [6]. Our approach to generate RegExs can be seen as a reverse abstraction process. We use a more abstract language that is compiled to RegExs instead of abstracting RegExs.

3 Constructive Alignment & YAPS

The concept of *Constructive Alignment* by Biggs [3] is related to constructivist learning theories, i.e., knowledge is seen to be *constructed* by each learner, as opposed to being *transferred* from teacher to learner. Thus, in order to facilitate successful learning a learning environment must be designed that allows the desired learning objectives (LOs) to be achieved through appropriate learning activities. The key to this is the definition of LOs at an appropriate level as well as the design of matching examination tasks. This is because, according to the principle of *testing drives learning*, the cognitive level of the examination tasks also determines the learning activities of the students. Thus, if what is to be learned is not reflected in the examination tasks, well-intentioned motivational teaching often fails due to a lack of student engagement.

To achieve higher cognitive learning objectives in an e-exam in engineering science, a technical environment is needed that offers much more than the construction of multiple choice tasks. Students should be able to analyze presented issues, to develop and justify independent solutions, and not just select predetermined answers. For such demanding examination scenarios that assess deeper understanding [13], the examination system *Your Open Examination System for Activating and emPowering Students* (YAPS) was developed at Hamburg University of Technology (TUHH), which allows for a more creative and

technically appropriate construction of examination tasks [1].

Several key design decisions were made in the development of YAPS: It is licensed open source and is cost-efficient for the university with regard to the required hardware resources. In Germany, there is no real alternative to self-hosting of the examination system used for reasons of data protection. This criterion alone excludes many candidates for other examination software. YAPS is characterized by a contactless and state-based operating concept that maps the testing procedure of TUHH, i.e., steps like having each student confirm they are fit to take the exam, checking student ID cards and whether they are actually registered, are supported by YAPS and can be performed while keeping a distance between each other (which was especially important during the COVID-19 pandemic, when exams at TUHH were still conducted in person). Finally, YAPS is designed to be extensible and built with well-known technologies (e.g., Typescript, Docker).

Other, similar softwares such as Autolab[2] [18], INGInious[3] [5] or ArTEMiS[4] [14] can also be self-hosted, are extensible, and the latter even supports Haskell out of the box. However, YAPS is already available at TUHH and well-integrated with the processes attached to conducting an exam. Its lack of support for Haskell can easily be remedied through its extensibility.

4 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 analyze our functional programming (FP) course and exams before the electronic exam (e-exam).

First, through describing and listing the teaching activities of the course, we capture its current state to identify any exam-related parts that may need to be updated. Then, we examine previous exams with regard to constructive alignment (CA) and task categories to identify potentials for automation and improvement of task design. Finally, having established candidates for change in both the teaching activities and in the exam, we devise a plan to reach our goal: an updated FP course with an e-exam that reduces effort through automation without reducing exam quality.

4.1 Course Description

The course is based on the textbook "Programming in Haskell" by Graham Hutton [9] and mainly targeted at first semester computer science students. The teaching activities of the course are aligned with the learning objectives (LOs) summarized in Table 1. Overall, there are three weekly activities: First, the lecture lays the foundation for the other activities. To a limited extent, it also features practical elements like executing code examples and short in-class exercises. 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, as it is common in other courses as well [2]. They can also help out each other and exchange different approaches towards solving the tasks. At the end of each lab session, the tutors discuss the solutions with each student, making sure that misunderstandings are detected and cleared up as early as possible. Third, there are homework exercise sheets that, in contrast to the labs, are supposed to be solved alone. They feature more complicated tasks that require and foster

[2] `https://github.com/autolab/Autolab`
[3] `https://github.com/UCL-INGI/INGInious`
[4] `https://github.com/ls1intum/Artemis`

Table 1: Learning objectives of our FP course as specified in the module handbook [7]

	Knowledge-based
K_1	Students apply principles, constructs, and simple design techniques of FP.
K_2	Students demonstrate ability to read Haskell programs, and explain Haskell syntax.
K_3	Students interpret warnings and find errors in programs.
K_4	Students apply fundamental data structures, data types, and type constructors
K_5	Students employ strategies for unit tests of functions and simple proof techniques for partial and total correctness.
K_6	Students distinguish laziness from other evaluation strategies.
	Skill-based
S_1	Students break a natural language description down in parts amenable to a formal specification and develop a functional program in a structured way.
S_2	Students assess different language constructs, make conscious selections both at specification and implementation level, and justify their choice.
S_3	Students analyze given programs and rewrite them in a controlled way.
S_4	Students design and implement unit tests and can assess the quality of their tests.
S_5	Students argue for correctness of their program.

a deeper understanding. To check their solutions for correctness, students upload their code files to an autograding system, so they can get feedback anytime. Usually the students need to provide a number of examples and tests for each function, and the given feedback reports on the correctness of functions, examples, and tests alike. For failing tests also the test input, expected and actual output are given. Finally, the solutions are discussed in a dedicated lecture hall exercise session.

During the last month of the lecture period, we introduce a few changes to what is described above, to help the students prepare for the exam. 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, so they get an idea of what to expect during the real one.

4.2 Existing Exams

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: 1. Types and Type Classes 2. List Comprehension 3. Pattern Matching 4. Recursion 5. Higher-order Functions 6. User-defined Types 7. Evaluation 8. Reasoning and Testing. 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. The results are summarized in Table 2. During the analysis we noticed that there are no warning messages in the exam, so the first part of K_3 is not checked at all.

Regarding the types of tasks, we found that all tasks can be presented in one of the following five categories: 1. Single choice: Select one option from two or more. 2. Multiple choice: Select zero or more options from two or more. 3. Snippet: Write a short piece of source code, e.g., a type, an expression, or even just a number. 4. Code: Write a larger piece of source code, e.g., a complete function or user-defined type. 5. Text: Write a text, e.g., some explanation or justification.

To make sure the e-exam assesses the same LOs as we did previously, we took the arguably most simple approach to design it: “translating” each task directly. With the first two categories we already

use task types that can directly be integrated into an e-exam. Integrating snippet and code tasks is more difficult, but the answers must follow the rules of Haskell by design, so at least partially automated assessment is possible for them. Text answers are comparatively unstructured, and students may answer in either English or German, which makes those tasks ineligible for automated assessment.

4.3 Consequences

The analysis of the teaching activities showed that the later lab sessions with recapitulation tasks are suited best for familiarizing the students with the e-exam format. Therefore, we decided to move those tasks to *Your Open Examination System for Activating and emPowering Students* (YAPS) instead of using text editor and terminal as usual. In addition, the exemplary exam that students can solve during the last lab appointment is also in electronic form. This way, all students have the opportunity to learn and try out how to use YAPS, and we can also test new ideas in a comparatively risk-free environment. Furthermore, our tutors are not only there to help out with any issues that may arise, but also to collect valuable feedback directly from the students, which we could not get any other way (especially observations on how the system is used and insights from student-to-student/tutor conversations). Through this feedback we can detect problems (e.g., task designs that are difficult to navigate) early and prevent them from occurring in the real exam.

Table 2: Learning objectives and types of existing exam tasks

Task	LOs	Type
1a	K_1, K_2, K_4	snippet
1b	K_1, K_2, K_4	multiple choice
2a	K_1, K_2, K_3	single choice, snippet
2b	K_1, K_2, S_2, S_3	code
3a	K_1, K_2, K_3, K_4	single choice, snippet
3b	K_1, K_2, K_4, S_3	code
4a	K_1, K_2	text
4b	K_1, K_2, S_1, S_2, S_3	code
5a	K_1, K_2	snippet
5b	K_1, K_2	text
5c	K_1, K_4, S_1, S_2	code
6a i	K_1, K_2, K_4, S_1	snippet
6a ii	K_1, K_2, K_3, K_4	single choice
6b	K_1, K_4, S_1	code
7a	K_1, K_2, K_6	text
7b	K_1, K_2, K_6	snippet
7c	K_1, K_2, K_6	text
8a	K_1, K_2, K_5, S_1, S_2, S_4	code
8b	K_1, K_2, K_5, S_5	text

Regarding the exam itself, we found ways to transform each of the task categories we identified to an e-exam. Single and multiple choice tasks can naturally remain. For snippet tasks we found that regular expressions (RegExs) can in all cases be used to detect correct solutions, 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. This is why we also allow for a manual assignment of points in case there was no match. Alternatively, snippet tasks could be checked using a compiler. While this does not solve the problem of grading partially correct solutions, it would reduce the workload because no RegExs need to be created. On the other hand, RegExs are arguably more flexible because they could also be used to detect common errors (by additionally specifying multiple erroneous RegExs), hence enabling awarding reduced amounts of points. Tasks that require writing a larger amount of source code can also remain, but additional work is required for the automated assessment here as well. In YAPS, programming tasks are worked on in an integrated code editor with syntax highlighting and the possibility to compile and execute the code, where the output of both compilation and execution

is shown to the students. For automated assessment we mainly employ randomized property tests via QuickCheck [4] and a new static analysis tool that provides information on used language features and functions. Welcome side effects of integrating a code editor are that programming tasks can now be executed in a more familiar way, and through error and warning messages from the compiler, LO K_3 is now fully included, while in the paper exam it was not. One important decision to make with automated programming tasks is whether only compiling code is accepted as an answer, or whether (partial) points are awarded also for non-compiling programs. We decided for the latter because we believe that a solution that is correct except for, e.g., a single wrongly-indented line, should still be worth points. Still, non-compiling code cannot be tested, so this reduces the degree of automation. Plain text tasks are the hardest to automate and there is no general way to do this. Therefore, we examined each task individually and found that there actually is a way to transform many of the text tasks in our exam in a similar way. We often ask students to make a decision and then justify it, so at least the decision part can be modeled as a single or multiple choice task, and only the reasoning is left for manual evaluation. 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.

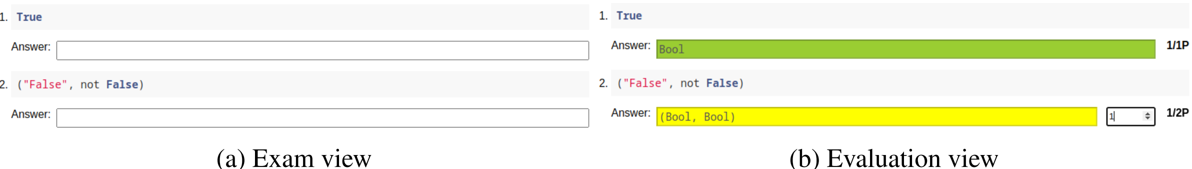
5 Realization

In the following we explain in detail how we realized the plan derived from the course analysis. First, we focus on the extensions to *Your Open Examination System for Activating and emPowering Students* (YAPS): Haskell compiler integration, regular expression (RegEx) tasks, proof puzzle evaluation, and the addition of a comment field for all tasks. Then, we introduce the code analysis tool for Haskell programming tasks and the RegEx generator program, together with its higher-level RegEx specification language.

5.1 Haskell Compiler Integration

To facilitate creating new programming tasks, we devised a template that provides a common 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 cannot modify this file, hence students cannot execute arbitrary code. They are limited to the functions evaluated in `main`, but we also control the types of these functions. Most importantly, these functions are never `IO` actions in our tasks, so we can leverage the type system of Haskell to prevent students from doing anything dangerous (the code is additionally isolated using containerization). 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*. Students cannot see this file. 5. Write the code to execute the tests, parse and interpret their output, and report the results back to YAPS into *evaluate.py*. Students cannot see this file.

This structure works for almost all tasks without modification, but is still flexible enough to allow for deviations. One example of that are tasks where students need to write tests themselves. Testing tests is more difficult than testing “normal” functions because we cannot write randomized property tests for them. Instead, we omit *main_test.hs* and provide two (or more) implementations of the function under



(a) Exam view (b) Evaluation view

Figure 1: Regular expression task

test: one that is correct, and one (or more) that is faulty. The code from *main.hs* is executed with the correct version during the exam. During evaluation, we link the student test with each of the versions of the function under test. Then we check whether the test passes for the correct version and fails for the faulty ones.

5.2 Checking Answers with Regular Expressions

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. The first answer is found to be correct. The second answer is only partially correct, so partial points are awarded through the numeric input field to the right. This input field is hidden when the RegEx matched to prevent any accidental modifications of the awarded points.

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 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.

5.3 New Algorithm to Evaluate Proof Puzzles

An example of the proof puzzle task type is given in Fig. 2. Students can drag the elements from right to left (and vice versa) to construct their proof. 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.

For clarity, we introduce the following notation: Let Σ^* be the set of all (Unicode character) strings. An *item* $(t,w) \in \Sigma^* \times \mathbb{Q}$ consists of the displayed text t and the associated weight w. Let $\mathbb{S}_I$ be the set of all sequences of items in $I \subseteq \Sigma^* \times \mathbb{Q}$. A *solution* $(s,p) \in \mathbb{S}_I \times \mathbb{Q}$ of length n consists of a sequence of items $s = \iota_1, \ldots, \iota_n$, and the associated number of points p. We denote the ith item of s as $\iota_i = (t_i, w_i)$.

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. The edit distance is defined as the weighted number of *insert* and *remove* operations required to transform one sequence of items into another. For an item $(_, w)$, the cost of inserting/removing it is equal to w. Algorithm 2 (Appendix) shows the complete procedure for calculating the number of points to award for a solution attempt. It can be summarized

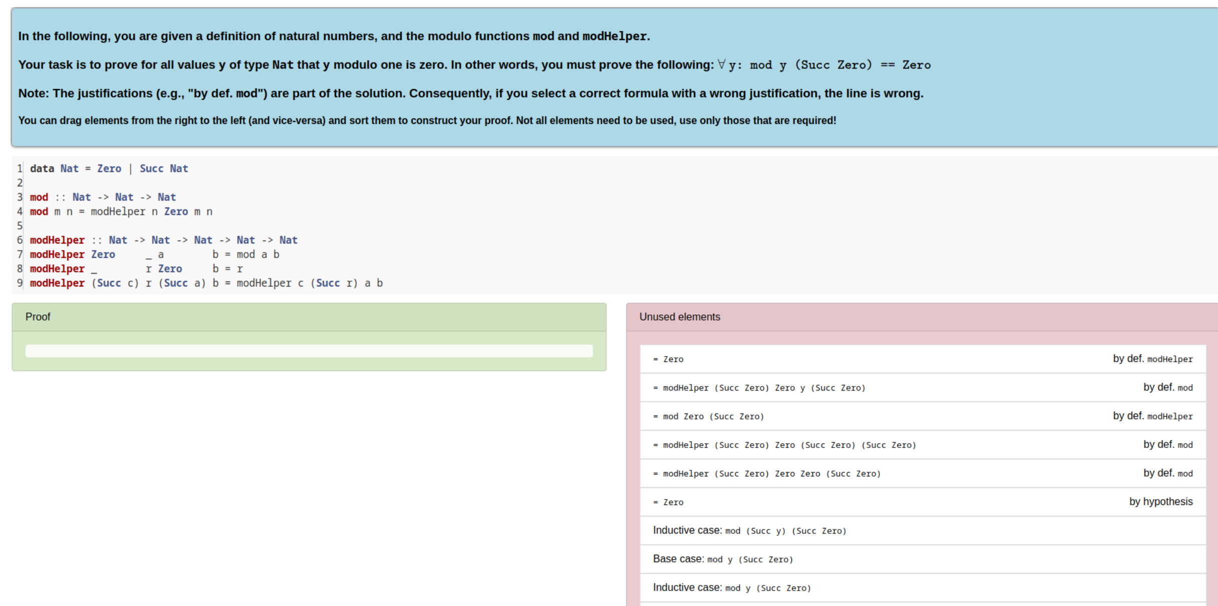


Figure 2: Proof puzzle task

as follows: For each possible solution, calculate the edit distance to the given solution attempt. Then, subtract the edit distance from the maximum amount of points that can be awarded for that solution. The resulting number of points is the maximum of these differences.

During testing, this algorithm produced reasonable results. Yet, after conducting the exam, we found that sometimes the resulting points were not in line with how we manually evaluated these tasks in the past, and that sometimes the results were even unfair, e.g., in the following case (cf. Figure 4, Appendix): Student A has correctly identified the inductive step of the proof and the next step, and gets 0.5 points. Student B has only correctly identified the base case and gets 2.0 points. 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. The new algorithm is given in Algorithm 1 and replaces lines 3 and 4 in Algorithm 2 (Appendix). It proceeds as follows: First, initialize the result with 0 (line 11), and look for the first entry point of a sequence in the given solution. The subroutine to search for the next sequence start (line 1) takes the current indices in the lists of solution and given answer items as input (both initially 0). It iterates over the solution sequence, and checks whether the current item is an entry point, and if it is also part of the given answer, a new synchronization point between solution and given answer is found. The subroutine returns the indices of the common item in the solution and in the given answer. If such a synchronization point cannot be found, the subroutine returns infinity for both indices, which terminates the main loop (line 13). The main loop iterates over the solution items as well, checking for each item whether it is still in the sequence or breaks it. If it is part of the sequence, its weight is added to the result, and the indices in the lists of solution and answer items are incremented. Otherwise, search for the next entry point and skip all items in between.

Algorithm 1 resolves the unfair grading explained above. Student A now gets 1.5 points, while

student B gets 1.0 (cf. Figure 5, Appendix). In case of proof by induction tasks, we usually specify the first line of the base case and the first line of the inductive step as entry points for sequences.

Algorithm 1 New proof puzzle evaluation

Input: $I \subset \Sigma^* \times \mathbb{Q}$ (items), $s \in \mathbb{S}_I \times \mathbb{Q}$ (solution), $a \in \mathbb{S}_I$ (solution attempt)
Output: Amount of points to award for a

```
1: function FindNextSequenceStart(solIdx, ansIdx)
2:     for solIdx < |s| do
3:         if IsSequenceStart(s_solIdx) and s_solIdx ∈ a then
4:             return (solIdx, index of s_solIdx in a)
5:         end if
6:         solIdx ← solIdx + 1
7:     end for
8:     return (∞, ∞)
9: end function
10:
11: r ← 0
12: (solIdx, ansIdx) ← FindNextSequenceStart(0, 0)
13: for solIdx < |s| and ansIdx < |s_a| do
14:     if s_solIdx = a_ansIdx then
15:         r ← r + w_solIdx
16:         (solIdx, ansIdx) ← (solIdx + 1, ansIdx + 1)
17:     else
18:         (solIdx, ansIdx) ← FindNextSequenceStart(solIdx, ansIdx)
19:     end if
20: end for
21: return r
```

5.4 Comment Field for Students

Another extension we added to the YAPS framework is a comment field for all exercises. We frequently found in paper exams that students add additional thoughts about exercises beyond their answers. Often these are explanations of how the exercise was understood or assumptions on which the answer of the student was based. While we aim to phrase all exercises in a way that neither an interpretation nor additional assumptions are necessary, we still wanted to give students the option to express these so that their answers can be evaluated in the right context. Therefore, we added an all-purpose text field at the bottom of each exercise with an explanatory text describing its purpose. Whatever is written into this field persists in addition to all actual answers so that we can read it during the correction of the exam. Another use case we found for this text field is that students wrote notes for themselves during the exam that they would ordinarily scribble in the margins of a paper exam.

5.5 Analyzing Student Code for Task-Relevant Features

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

Listing 1: Example of quicksort in Haskell[7]

```
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
  where lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Listing 2: Code analyzer output for Listing 1

```
{ "functions": [{
    "name": "quicksort",
    "patMatch": true,
    "guards": false,
    "listComprehension": false,
    "hasIf": false,
    "hasCase": false,
    "args": [ "p", "xs" ],
    "calledFns": [ "quicksort", "++", "filter", "<", ">=" ],
    "declaredFns": [ ]
}]}
```

we devised a new tool[5] that itself is written in Haskell. It takes a source code file as input, and outputs certain information on each function, that is defined in the input file, in JavaScript Object Notation (JSON). For example, with the well-known implementation of quicksort, given in Listing 1, the program outputs the JSON shown in Listing 2. 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 extract that information, we use the `haskell-src`[6] package to build and traverse the abstract syntax tree (AST) of the given source file. The program itself is split into a library that provides the described functionality, and an application that uses this library. Altogether, the entire program consists of less than 500 lines of code of which most are used for pattern matching on the constructors of the sum types that represent the AST and advancing the search in depth-first manner.

The `haskell-src` library parses the given file to the `HsModule` type which features a list of declarations. We filter those declarations for functions (`HsFunBind`) and constants (`HsPatBind`), and then check for the features and functions used. The `HsFunBind` constructor holds the clauses of the function (`[HsMatch]`), and the clauses provide access to guards (`HsGuardedRhss`), as well as access to the patterns used in the left hand side. Any pattern that is not a name (`HsPVar`) is considered a use of pattern matching. Names, however, are added to the list of arguments (subpatterns are also considered, e.g., `(x:xs)` is considered a use of pattern matching and adds the arguments `x` and `xs`). To collect the remaining information, the right hand sides (`HsRhs`) and local declarations (`HsDecl`) are traversed similarly.

---

[5] `https://collaborating.tuhh.de/cda7728/check-hs-task-restrictions`
[6] `https://hackage.haskell.org/package/haskell-src`
[7] `https://wiki.haskell.org/Introduction#Quicksort_in_Haskell`

Listing 3: Example RegEx generator output

```
^\s*(?:\((?=(?:[^\(\)]*)\)))?\s*Num\s*\s(?<a>[_a-z][_a-zA-Z0-9']*)\s
 ↪ *(?:(?<=\((?:[^\(\)]*))\))?\s*=>\s*\[\s*\k<a>\s*\]\s*->\s*(?:
 ↪ String\s*|\[\s*Char\s*\]\s*)$
```

5.6 Generating Flexible Regular Expressions

To increase automation during exam creation we developed a second tool[8] 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. 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. The result is shown in Listing 3 and illustrates how large these expressions can become. RegExs of that size are difficult to understand for humans, so writing them by hand is rather error-prone.

Therefore, the HTRSL provides an additional layer of abstraction tailored to common patterns in Haskell expressions. Its grammar is given in Listing 4 (Appendix) in labelled Backus-Naur form: A HTRSL file is a semicolon-separated list of specifications. Each specification is a description. A description is a list of description items. Allowed items are:

- Literals: Match the string given in double-quotes literally.
- Identifiers: Match any identifier (`[_a-z][_a-zA-Z0-9']*`). If reused in the same description, only matches the same identifier that was found at the first occurrence.
- Mandatory whitespace: Match any whitespace (`\s+`), represented as `\\` in the description.
- Alternatives: Match any of the alternatives that are given in square brackets and separated by `|`.
- Optional parentheses: Match what is specified inside the parentheses, either surrounded by parentheses or not. Cannot be nested.

We use the BNF Converter[9] to generate a parser for the language. This way, we obtain an abstract syntax tree that we can traverse to generate the desired RegExs.

In the future, we plan to add support for automatically testing the generated RegExs. Currently, this is done manually by testing each RegEx for some positive and negative examples. This can be automated as well by adding the test strings to the specification, so the tool can directly check whether the generated RegEx matches all positive examples, and rejects all negative ones.

6 Evaluation

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

[8] https://collaborating.tuhh.de/cda7728/gen-hs-task-regexs

[9] https://hackage.haskell.org/package/BNFC

targeted at evaluating how far we have come with respect to our initial motivation: reducing the effort of conducting an exam through automation. We believe that, by design, we did not sacrifice constructive alignment (CA), yet CA compliance comes at the cost of less automation, and we wanted to know how high that cost actually is. While reducing the effort of conducting the exam mostly benefits us, the examiners, we also wanted to provide the students with a more comfortable and familiar way of taking an exam, especially when it comes to programming tasks. Finally, we reflect on our own experiences, from creating the exam to grading it, to answer the last question.

6.1 Automated Grading

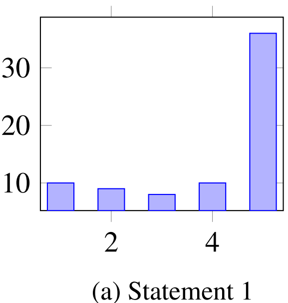
To evaluate the degree of automation of the grading process, we divide all tasks in the e-exam into the following categories: 1. Fully automated: human intervention is not required in any case. 2. Automated if correct: human intervention is only required if the given answer was incorrect (to potentially 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. The results are given in Table 3 for an example exam (we do not rule out that other exams have, e.g., a task 1c). They show a direct mapping between task type and category: multiple/single choice or proof puzzle → fully automated, regular expression (RegEx) or Programming → automated if correct, text → not automated. When two task types are combined, the resulting degree of automation is the one that is worse. Combinations of text tasks with one that is at least "automated if correct" result in a "partially automated task". We see that most tasks are "automated if correct", which is our second-best category. Yet, just from the categorization we cannot determine how much this reduces the effort required for grading the exam in practice. Still, all tasks except for one feature at least some degree of automation, and overall that makes it likely to reduce the required efforts. In other words, the results can be summarized as follows: Correct solutions can in most cases be awarded the full amount of points automatically, but awarding partial points is difficult.

Table 3: Categorization of tasks by degree of automation

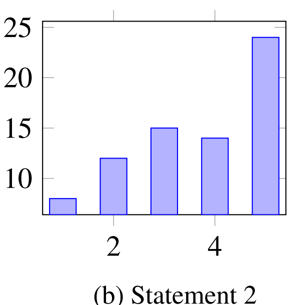
Task	Task Type	Category
1a	RegEx	Automated if correct
1b	Multiple Choice	Fully automated
2a	Single choice + RegEx	Automated if correct
2b	Programming	Automated if correct
3a	Single Choice + RegEx	Automated if correct
3b	Programming	Automated if correct
4a	Single Choice + Text	Partially automated
4b	Programming	Automated if correct
5a	Programming	Automated if correct
5b	Text	Not automated
5c	Programming	Automated if correct
6a	Programming	Automated if correct
6b	Multiple Choice	Automated
6c	Programming	Automated if correct
7a	Single Choice + Text	Partially automated
7b	RegEx	Automated if correct
7c	Multiple Choice + Text	Partially automated
8a	Programming	Automated if correct
8b	Proof Puzzle	Fully automated

6.2 Student View

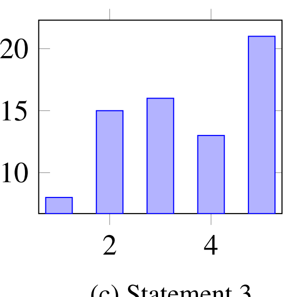
To capture the view of the students, we conducted a short poll immediately after the exam. When time for the exam was up, the exam browser automatically redirected the students to the poll website. On the one hand this allowed us to capture the opinion of the students without delay and reduced external influence.



(a) Statement 1



(b) Statement 2



(c) Statement 3

Figure 3: Poll results
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.

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.

Of the 77 exam participants, 73 answered the poll. The results are shown in Fig. 3. Figure 3a shows that the majority of the participants ($36+10=46 \equiv 63\%$) generally agrees that an e-exam is the right examination format for the course. Most ($24+14=38 \equiv 52\%$) also found working with the integrated compiler natural, yet comparatively many students are neutral on this question ($15 \equiv 21\%$), as shown in Fig. 3b. For the third question, Fig. 3c shows that less than half of the participants ($21+13=34 \equiv 45\%$) agreed that they can express their thoughts as they could in a paper exam. Almost a third of the students ($8+15=23 \equiv 32\%$) disagrees. 29 students even provided additional feedback that ranges from constructive criticism to outright ecstasy ("It was SIIIICKKK!!! Also perfect amount of time for the tasks. I finished in the last 5 seconds"). Other critical remarks were:

- "It's sometimes a little bit complicated for color-blind people to see whether I marked a task already as green or it's still orange." In *Your Open Examination System for Activating and em-Powering Students* (YAPS), students can mark tasks green when they finished them, unseen tasks are gray, and tasks where some input was given appear orange. This choice of colors apparently is problematic for some and should be changed.

- "Maybe it would be better to make more friendly function names" Sometimes the function names are a very short abbreviation that at first sight may appear as a random sequence of letters. Naming functions this way is a habit that stems from creating paper exams where horizontal space for code is much more limited than in the YAPS code editor, so we should consider using the additional space and use more descriptive function names.

- "In my opinion it's nicer to write the proof (last task) as text instead of drag and drop, as the drag and drop fields all look very similar, and it's more like a task to find the correct field, than to actually think about the proof" This is a hint that the proof puzzle may not be the optimal way to assess the ability of the students to construct a proof, and we should investigate other ways.

- "Please get new computer mice. That is really necessary" (translated from German). When designing the e-exam we did not think about hardware, apart from how the screen size impacts the

visual presentation of the exam, as it was provided by our university anyway. This statement shows that hardware does play a role and should be taken into account.

- "C and G are difficult to differentiate on the small laptop screen" (translated from German). This goes into a similar direction as the previous statement. While we checked how all tasks look on the laptop screens that are used during the exam, we did not see this problem. Possibly using a different font can help here.

Overall we think the results from the poll are encouraging, but there are some details that need improvement. Especially the free text student feedback shows how we can improve the exam, and that it is important to incorporate student feedback when designing and improving an e-exam.

6.3 Examiner View

In this section we report our own experiences with the complete e-exam workflow from creation to grading, and highlight successes as well as difficulties.

When creating a new e-exam, we begin with writing the tasks and solutions outside YAPS. Where possible, we also write tests for the solutions to ensure they are indeed correct. During that phase we employ literate Haskell with Markdown through `markdown-unlit`[10] to combine text, code, and tests in the same file. This workflow has two main advantages: we can render the exam in PDF format and split between a PDF that contains all tasks, and one that contains the solutions, and we can directly execute the exam code and tests. For a paper exam we would instead write the exam in LaTeX(with tests in a separate file), which can be more time-intensive because we need to pay more attention to the resulting layout.

When the exam tasks themselves are finalized, we implement them in YAPS. The task texts can mostly be simply pasted into Hypertext Markup Language (HTML) templates, answers for single or multiple choice tasks are easily configured in JavaScript Object Notation (JSON) files, and RegExs are generated automatically by our tool. This part of implementing the exam is a rather tedious process, but the templates and tool support help to minimize redundancies. Yet, we manually perform the same steps multiple times, which likely could be automated. More creative work is required for proof and programming tasks. Here, we need to find sensible distractors (proof tasks) and write suitable property tests (programming tasks). This is the most time-consuming part of the entire exam creation process. Finally, we perform a few (< 5) rounds of testing the finished exam to make absolutely sure everything works as intended. For testing, we mostly enter correct answers and check whether they are detected as such. Usually there are few mistakes, e.g., typos, but sometimes also problems with the tests for programming tasks occur.

On the day of the exam, the required technical infrastructure is already set up by a team from our university [28]. On the other hand, conducting an e-exam may require more organizational overhead because we only have 100 laptops available, so for large exams we need to do multiple rounds. This introduces a new set of difficulties because we need to make sure no information about the exam is given from, e.g., the first round to the second. One way to do this is to provide different exams, which increases the required effort substantially. Here, the randomization features of YAPS are very useful to create task variations with little effort. When time for the exam is up, YAPS prevents any further input from students, and collects all given answers (or rather collects them continuously during the exam to make sure no data is lost). We can then trigger the automated evaluation, which may take some minutes, so we come back later.

[10] `https://github.com/sol/markdown-unlit`

Grading is then done in YAPS as well. We almost exclusively look at those tasks where mistakes were found by the automated grading to manually distribute points. Additionally, we sample a few answers detected as correct to check whether the automated grading worked as intended, which usually is the case. If there is a mistake, however, we can correct it and execute the automated evaluation again. Grading an exam with 77 participants took a single person approximately two work days. The last paper exam with 136 participants took two people approximately three work days. Unfortunately, "grading effort" is hard to measure, and the numbers are difficult to compare because of the very different numbers of participants. Still, it is a good sign that the exam could be graded by a single person in a comparatively short amount of time. Assuming that with twice as many participants (154) the required amount of time would double as well, a single person could grade as many e-exams as two people who grade the same exam in paper, in the same timeframe.

Table 4: Comparison of paper exam (fall 2020) and e-exam (fall 2021) results

Task	Paper Exam		E-Exam	
	max	avg	max	avg
1a	7	4	6	4
1b	5	3	6	3
2a	6	4	6	4
2b	6	3	6	3
3a	10	6	10	6
3b	3	1	3	2
4a	6	4	6	4
4b	4	3	4	2
5a	3	1	3	2
5b	5	1	5	1
5c	5	1	5	1
6a	6	3	6	3
6b	6	2	6	4
7a	6	4	6	4
7b	6	3	6	4
7c	5	1	5	1
8a	5	2	6	1
8b	6	2	5	2

Table 4 shows the results per task for our last paper exam, and the e-exam we report on in this paper. For each task, the maximum achievable number of points, and the average amount of points scored by students are shown. Rows with better/worse results (in comparison to the paper exam) are colored blue/red. Overall, the students performed very similarly. The few, minor differences we see cannot be clearly attributed to a specific phenomenon. For instance, tasks 2b, 3b, 4b, 5a, 5c, 6b, and 8a are programming tasks, but the students sometimes performed better, worse, or the same. When deciding for the *Proof Puzzle*, we thought that the proof task (8b) may become noticeably easier because in the paper exam the proof had to be created entirely from scratch. However, the results suggest that in this exam creating the proof was similar in difficulty

In summary, creating an e-exam takes more time than creating a paper exam, but with maturing templates and tools we can confidently expect that this overhead becomes smaller. Conducting and grading, however, takes less time now. The exam results were almost exactly the same compared to a previous paper exam. Altogether, we conclude that the approach to create an e-exam from a paper exam by direct "translation" of each task was successful.

7 Summary & Future Work

We showed how a traditional paper exam with complex proof and programming tasks can be transformed to an electronic exam (e-exam) with automated grading, but without sacrificing constructive alignment (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 regular expressions (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 task 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 the initial literate Haskell Markdown file that contains the task texts, solutions, and point distribution.

References

[1] F. H. Bahnsen & G. Fey (2021): *YAPS - Your Open Examination System for Activating and emPowering Students*. In: *2021 16th Int. Conf. Comput. Sci. Educ. (ICCSE)*, pp. 98–103, doi:10.1109/ICCSE51940.2021.9569549.

[2] A. Bieniusa, M. Degen, P. Heidegger, P. Thiemann, S. Wehr, M. Gasbichler, M. Sperber, M. Crestani, H. Klaeren & E. Knauel (2008): *HtDP and DMdA in the Battlefield: A Case Study in First-Year Programming Instruction*. In: *FDPE '08: Proc. 2008 Int. Workshop Funct. Declar. Program. Educ.*, pp. 1–12, doi:10.1145/1411260.1411262.

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

[4] K. Claessen & J. Hughes (2000): *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In: *ICFP '00: Proc. Fifth ACM SIGPLAN Int. Conf. Funct. Program.*, pp. 268–279, doi:10.1145/351240.351266.

[5] G. Derval, A. Gego, P. Reinbold, B. Frantzen & P. Van Roy (2015): *Automatic Grading of Programming Exercises in a MOOC Using the INGInious Platform*. In: *Proc. Eur. MOOC Stakehold. Summit 2015*, pp. 86–91.

[6] M. Erwig & R. Gopinath (2012): *Explanations for Regular Expressions*. In: *Fundam. Approaches Softw. Eng.*, *Lect. Notes Comput. Sci.* 7212, pp. 394–408, doi:10.1007/978-3-642-28872-2_27.

[7] Hamburg University of Technology (2022): *Module Manual Bachelor of Science (B. Sc.) Computer Science – Cohort: Winter Term 2021*. Available at `https://studienplaene.tuhh.de/po/E/mhb_CSBS_kh_w21_von_20220524_v_0_en.pdf`.

[8] J. Hollingsworth (1960): *Automatic Graders for Programming Classes*. *Comm. ACM* 3(10), pp. 528–529, doi:10.1145/367415.367422.

[9] G. Hutton (2016): *Programming in Haskell*, second edition. Cambridge University Press, doi:10.1017/CBO9781316784099.

[10] F. K. Jacobsen & J. Villadsen (2023): *On Exams with the Isabelle Proof Assistant*. In: *Proc. 11th Int. Workshop Theorem Proving Compon. Educ. Softw.*, *Electron. Proc. Theor. Comput. Sci.* 375, pp. 63–76, doi:10.4204/EPTCS.375.6.

[11] J. C. S. Kadupitiya, S. Ranathunga & G. Dias (2016): *Automated Assessment of Multi-Step Answers for Mathematical Word Problems*. In: *2016 Sixt. Int. Conf. Adv. ICT Emerg. Reg. (ICTer)*, pp. 66–71, doi:10.1109/ICTER.2016.7829900.

[12] K. Kappelmann, J. Rädle & L. Stevens (2022): *Engaging, Large-Scale Functional Programming Education in Physical and Virtual Space*. In: *Proc. Tenth Elev. Int. Workshop Trends Funct. Program. Educ.*, *Electron. Proc. Theor. Comput. Sci.* 363, pp. 93–113, doi:10.4204/EPTCS.363.6.

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

[14] S. Krusche & A. Seitz (2018): *ArTEMiS: An Automatic Assessment Management System for Interactive Learning*. In: *SIGCSE'18: Proc. 49th ACM Tech. Symp. Comput. Sci. Educ.*, pp. 284–289, doi:10.1145/3159450.3159602.

[15] M. Kuikka, M. Kitola & M.-J. Laakso (2014): *Challenges When Introducing Electronic Exam*. *Res. Learn. Technol.* 22, doi:10.3402/rlt.v22.22817.

[16] H. Loftsson & Á. Matthíasdóttir (2021): *Moving Classes in a Large Programming Course Online: An Experience Report*. In: *Second Int. Comput. Program. Educ. Conf. (ICPEC 2021), Open Access Ser. Inform. (OASIcs)* 91, pp. 2:1–2:13, doi:10.4230/OASIcs.ICPEC.2021.2.

[17] M. McCartin-Lim, B. Woolf & A. McGregor (2018): *Connect the Dots to Prove It: A Novel Way to Learn Proof Construction*. In: *SIGCSE'18: Proc. 49th ACM Tech. Symp. Comput. Sci. Educ.*, pp. 533–538, doi:10.1145/3159450.3159609.

[18] D. Milojicic (2011): *Autograding in the Cloud: Interview with David O'Hallaron*. *IEEE Internet Comput.* 15(1), pp. 9–12, doi:10.1109/MIC.2011.2.

[19] D. S. Morris (2003): *Automatic Grading of Student's Programming Assignments: An Interactive Process and Suite of Programs*. In: *33rd Annu. Front. Educ.*, 3, pp. S3F–1 – S3F–6, doi:10.1109/FIE.2003.1265998.

[20] T. Nipkow, M. Wenzel & L. C. Paulson (2002): *Isabelle/HOL*. *Lect. Notes Comput.Sci.* 2283, doi:10.1007/3-540-45949-9.

[21] J. C. Paiva, J. P. Leal & Á. Figueira (2022): *Automated Assessment in Computer Science Education: A State-of-the-Art Review*. *ACM Trans. Comput. Educ.* 22(3), pp. 34:1–34:40, doi:10.1145/3513140.

[22] V. Pieterse (2013): *Automated Assessment of Programming Assignments*. In: *CSERC'13: Proc. 3rd Comput. Sci. Educ. Res. Conf.*, pp. 45–56.

[23] S. Poulsen, S. Kulkarni, G. Herman & M. West (2022): *Efficient Partial Credit Grading of Proof Blocks Problems*, doi:10.48550/arXiv.2204.04196.

[24] S. Poulsen, M. Viswanathan, G. L. Herman & M. West (2022): *Proof Blocks: Autogradable Scaffolding Activities for Learning to Write Proofs*. In: *ITiCSE '22: Proc. 27th ACM Conf. Innov. Technol. Comput. Sci. Educ. Vol. 1*, pp. 428–434, doi:10.1145/3502718.3524774.

[25] T. Rajala, E. Kaila, R. Lindén, E. Kurvinen, E. Lokkila, M.-J. Laakso & T. Salakoski (2016): *Automatically Assessed Electronic Exams in Programming Courses*. In: *ACSW'16: Proc. Australas. Comput. Sci. Week Multiconf.*, pp. 1–8, doi:10.1145/2843043.2843062.

[26] J. Schneider, R. Richner & M. Riser (2022): *Towards Trustworthy AutoGrading of Short, Multi-lingual, Multi-type Answers*. *Int. J. Artif. Intell. Educ.* 33(1), pp. 88–118, doi:10.1007/s40593-022-00289-z.

[27] J. Sheard, Simon, A. Carbone, D. Chinn, M.-J. Laakso, T. Clear, M. de Raadt, D. D'Souza, J. Harland, R. Lister, A. Philpott & G. Warburton (2011): *Exploring Programming Assessment Instruments: A Classification Scheme for Examination Questions*. In: *ICER'11: Proc. Seventh Int. Workshop Comput. Educ. Res.*, pp. 33–38, doi:10.1145/2016911.2016920.

[28] D. Sitzmann, K. Kruse, D. Gallaun, N. Kubick, B. Reinhold, M. Schnabel, L. Thoms, H. Barbas & D. Meiling (2022): *Aufbau eines mobilen Testcenters für die Hamburger Hochschulen im Rahmen des Projekts MINTFIT E-Assessment*. *Hochschullehre* 8, pp. 113–129, doi:10.3278/HSL2208W.

[29] C. Stergiopoulos, P. Tsiakas, D. Triantis & M. Kaitsa (2006): *Evaluating Electronic Examination Methods Applied to Students of Electronics. Effectiveness and Comparison to the Paper-and-Pencil Method.* In: *IEEE Int. Conf. Sens. Netw. Ubiquitous Trust. Comput. (SUTC'06)*, 2, pp. 143–151, doi:10.1109/SUTC.2006.65.

[30] L. C. Ureel II & C. Wallace (2019): *Automated Critique of Early Programming Antipatterns*. In: *SIGCSE '19: Proc. 50th ACM Tech. Symp. Comput. Sci. Educ.*, pp. 738–744, doi:10.1145/3287324.3287463.

[31] L.-X. Zheng, S. Ma, Z.-X. Chen & X.-Y. Luo (2021): *Ensuring the Correctness of Regular Expressions: A Review*. *Int. J. Autom. Comput.* 18(4), pp. 521–535, doi:10.1007/s11633-021-1301-4.

Appendix

Ihre Antwort		Lösung		Punkte
				5.00
		Base case: `grt (add Zero (Succ x)) Zero`		-1
		`= True`	by def. `grt`	-1
Base case: `grt (add n (Succ Zero)) n`				0
Inductive step: `grt (add (Succ n) (Succ x)) (Succ n)`		Inductive step: `grt (add (Succ n) (Succ x)) (Succ n)`		0
`= grt (Succ (add n (Succ x))) (Succ n)`	by def. `add`	`= grt (Succ (add n (Succ x))) (Succ n)`	by def. `add`	0
		`= grt (add n (Succ x)) n`	by def. `grt`	-0.5
		`= True`	by hyp.	-1
`= grt (Succ (add x n)) n`	by def. `add`			0
`= grt (Succ (add Zero n)) n`	by def. `add`			0
`= grt (Succ n) n`	by def. `add`			0
`= True`	by def. `grt`			-1
				Total: 0.50

(a) Student A

Ihre Antwort		Lösung		Punkte
				5.00
Base case: `grt (add Zero (Succ x)) Zero`		Base case: `grt (add Zero (Succ x)) Zero`		0
`= grt (Succ x) Zero`	by def. `grt`			0
Inductive step: `grt (add (Succ Zero) (Succ x)) (Succ Zero)`				0
`= grt (Succ (Succ x)) (Succ Zero)`	by def. `add`			0
`= True`	by def. `grt`	`= True`	by def. `grt`	0
		Inductive step: `grt (add (Succ n) (Succ x)) (Succ n)`		-1
		`= grt (Succ (add n (Succ x))) (Succ n)`	by def. `add`	-0.5
		`= grt (add n (Succ x)) n`	by def. `grt`	-0.5
		`= True`	by hyp.	-1
				Total: 2.00

(b) Student B

Figure 4: Example of unfair grading with old algorithm
Left column: student solution, middle column: our solution, right column: points

Ihre Antwort	Lösung	Punkte
		5.00
Base case: `grt (add n (Succ Zero)) n`	Base case: `grt (add Zero (Succ x)) Zero`	0.00
	`= True` by def. `grt`	0.00
Inductive step: `grt (add (Succ n) (Succ x)) (Succ n)`	Inductive step: `grt (add (Succ n) (Succ x)) (Succ n)`	1.00
`= grt (Succ (add n (Succ x))) (Succ n)` by def. `add`	`= grt (Succ (add n (Succ x))) (Succ n)` by def. `add`	0.50
`= grt (Succ (add x n)) n` by def. `add`	`= grt (add n (Succ x)) n` by def. `grt`	0.00
`= grt (Succ (add Zero n)) n` by def. `add`	`= True` by hyp.	0.00
`= grt (Succ n) n` by def. `add`		0.00
`= True` by def. `grt`		0.00
		Total: 1.50

(a) Student A

Ihre Antwort	Lösung	Punkte
		5.00
Base case: `grt (add Zero (Succ x)) Zero`	Base case: `grt (add Zero (Succ x)) Zero`	1.00
`= grt (Succ x) Zero` by def. `grt`	`= True` by def. `grt`	0.00
Inductive step: `grt (add (Succ Zero) (Succ x)) (Succ Zero)`	Inductive step: `grt (add (Succ n) (Succ x)) (Succ n)`	0.00
`= grt (Succ (Succ x)) (Succ Zero)` by def. `add`	`= grt (Succ (add n (Succ x))) (Succ n)` by def. `add`	0.00
`= True` by def. `grt`	`= grt (add n (Succ x)) n` by def. `grt`	0.00
	`= True` by hyp.	0.00
		Total: 1.00

(b) Student B

Figure 5: No unfair grading with new algorithm
Left column: student solution, middle column: our solution, right column: points

Algorithm 2 Default proof puzzle evaluation

Input: $I \subset \Sigma^* \times \mathbb{Q}$ (items), $S \subset \mathbb{S}_I \times \mathbb{Q}$ (solutions), $a \in \mathbb{S}_I$ (solution attempt)
Output: Amount of points to award for a

1: $r \leftarrow 0$
2: **for all** $(s_i, p_i) \in S$ **do**
3: $\quad d \leftarrow \textsc{EditDistance}(s_i, a)$
4: $\quad r_i \leftarrow \max\{0, p_i - d\}$
5: $\quad$ **if** $r_i > r$ **then**
6: $\qquad r \leftarrow r_i$
7: $\quad$ **end if**
8: **end for**
9: **return** r

Listing 4: Haskell Task RegEx Specification Language grammar

```
entrypoints [Spec] ;
separator Spec ";" ;

SNoTests.    Spec ::= Desc ;

D. Desc ::= [DescItem] ;
separator nonempty DescItem "" ;

DLit.    NoParensDescItem ::= String ;
DName.   NoParensDescItem ::= Ident ;
DSpace.  NoParensDescItem ::= "\\" ;
DNAlt.   NoParensDescItem ::= "[" [NoParensDescAlt] "]" ;
DND.     DescItem ::= NoParensDescItem ;
DPar.    DescItem ::= "(" [NoParensDescItem] ")" ;

separator nonempty NoParensDescItem "" ;

DNAltElem. NoParensDescAlt ::= [NoParensDescItem] ;
separator nonempty NoParensDescAlt "|" ;

comment "--";
comment "{-" "-}";
```

Regular Expressions in a CS Formal Languages Course

Marco T. Morazán
Seton Hall University
morazanm@shu.edu

Regular expressions in an Automata Theory and Formal Languages course are mostly treated as a theoretical topic. That is, to some degree their mathematical properties and their role to describe languages is discussed. This approach fails to capture the interest of most Computer Science students. It is a missed opportunity to engage Computer Science students that are far more motivated by practical applications of theory. To this end, regular expressions may be discussed as the description of an algorithm to generate words in a language that is easily programmed. This article describes a programming-based methodology to introduce students to regular expressions in an Automata Theory and Formal Languages course. The language of instruction is FSM in which there is a regular expression type. Thus, facilitating the study of regular expressions and of algorithms based on regular expressions.

1 Introduction

Historically, formal languages and automata theory courses are theoretical pencil-and-paper courses. Students design algorithms in theory (i.e., without implementing them) and write theorems based on the algorithms they design. If there is a bug in the algorithm then it is commonly the case (especially among students) that the bug is not discovered and there is, of course, also a bug in the proof of a theorem. This truly goes against the grain of a Computer Science education. Computer Science students are trained to design and implement algorithms. Unit testing and runtime bugs give them immediate feedback providing the opportunity to make corrections before submitting work for grading. This is rather difficult to do if algorithms are only designed and never implemented, because few instructors, if any, have time to provide feedback on draft solutions.

Regular expressions are introduced as a finite language-representation. Such a representation is needed, because many interesting languages are not finite. That is, they contain an infinite number of words and, therefore, it is impossible to list all the words in the language. A finite representation for a language, of course, must be written using a finite number of symbols and must be different from the representation used for any other language. If Σ is an alphabet used to write the finite representations of languages then all possible finite language representations are in Σ^*. This means that the language of finite language representations is countably infinite (i.e., they can be printed in alphabetical order like the words in a complete English dictionary).

Before writing their first regular expression, an early lesson students learn is that, 2^{Σ^*}, Σ^*'s power set is uncountable. So, there is a countable number of finite language representations and an uncountable number of languages to represent. Therefore, a finite representation for each language does not exist. The best that we can achieve is to develop a finite representation for some interesting languages. As long as a representation is finite the majority of languages cannot be represented. Although it is important for students to understand this result, the tone set by this approach is one that is usually found too theoretical by Computer Science students in a Formal Languages and Automata Theory course. This leads to apathy towards the material. This apathy is an unfortunate side-effect because regular expressions are about

Elena Machkasova (Ed.): International Workshop on Trends
in Functional Programming in Education 2023 (TFPIE 2023)
EPTCS 382, 2023, pp. 45–63, doi:10.4204/EPTCS.382.3

programming and algorithms–a fact lost by most students studying regular expressions for the first time in a formal languages course.

This article outlines a programming-based approach to teaching students about regular expressions in their first automata theory course using `FSM` (*F*inite *S*tate *M*achines). `FSM` is a domain-specific language in which regular expressions, state machines, and grammars are types [10]. Students are exposed to all the theory addressed by a traditional non-programming-based automata theory course, but are engaged by programming regular expressions and by designing/implementing programs based on regular expressions. The climax of this module brings students to the realization that regular expressions are an elegant way to describe an algorithm for generating members of a language. This module is covered before finite-state automatons are discussed in a course for third- and fourth-year undergraduate students at Seton Hall University.

The article is organized as follows. Section 2 reviews and contrasts approaches to teaching students about regular expressions. Section 3 briefly outlines regular expressions in `FSM`. Section 4 discusses classroom examples of programming with regular expressions. Section 5 discusses how students implement a word-generating function given a regular expression. Section 6 discusses applications of regular expressions and outlines password generation using regular expressions. Finally, Section 7 presents concluding remarks and directions for future work.

2 Related Work

There is wide-ranging treatment of regular expressions in Formal Languages and Automata Theory textbooks. Most textbooks start with finite-state automatons and discussion leads to regular expressions (e.g., [3, 5, 8, 12, 13]). Other textbooks start with regular expressions and then move to finite-state automatons (e.g., [4]). Regardless of when regular expressions are introduced, the depth of their treatment varies a great deal. Some textbooks only provide an informal definition for regular expressions, briefly discuss an application (e.g., lexical analysis), and quickly move to the equivalence between regular expressions and finite-state automatons (e.g., [8]). Most textbooks go a little further providing a formal definition and a set of examples before moving on to the equivalence between regular expressions and finite-state automatons (e.g., [3, 5]). The work presented in this article is delivered before finite-state automatons are presented. It provides students with a formal definition and regular expression examples. In contrast, however, the formal definition is as a type instance in a programming language and the examples are executable programs.

Sipser [13] and Lewis and Papadimitriou [4] have a more in-depth treatment of regular expressions. They motivate regular expressions as a finite representation that may be used to describe infinite languages. They provide a formal definition, examples, and discuss properties of regular expressions. Sipser briefly discusses identity properties and then lexical analysis before moving to the equivalence of regular expressions and finite-state automatons. Lewis and Papadimitriou focus a bit less on mathematical properties and informally discuss how regular expressions outline the steps to generate words in the language they describe. They reject, however, calling these steps an algorithm because of their nondeterministic nature (e.g., choosing the number of repetitions when generating a word for a Kleene star regular expression). In a similar manner, the work presented in this article presents students with a formal definition and examples. In contrast, mathematical properties of regular expressions are less emphasized and examples purposely lead to an algorithm and its implementation for generating words in a regular expression's language. The algorithm fully recognizes that randomness (i.e., nondeterminism) has its role in computation. As pointed out by Lewis and Papadimitriou, the result of generating a word

is not predictable. This, however, is only part of the story. There are properties that every word generated, given a regular expression, must satisfy. Therefore, unit tests using property-based testing may be written to validate any generated word.

Rich [12] discusses mathematical properties and several applications of regular expressions (e.g., lexical analysis, spam filtering, and password validity). Of the classical Formal Languages and Automata Theory textbooks, this is the most algorithmic. It emphasizes algorithms based on regular expressions, but only presents them as pseudocode. There is a discussion addressing the generation of words in the language of a given regular expression. For example, Rich suggests to *think of any expression that is enclosed in a Kleene star as a loop that can be executed zero or more times*. As Rich, the work presented in this article focuses on algorithms. In contrast, however, the work presented in this article also focuses on implementation. A word-generating function is fully implemented based on the experience students gain from implementing regular expressions. Students walk away understanding how to design and implement a word-generating function for any given regular expression.

3 A Brief Introduction to Regular Expressions in `FSM`

`FSM` is a domain-specific language embedded in `Racket` [2] and its use is specified as follows: `#lang fsm`. It inherits from `Racket` its syntax and its rich set of primitive functions. It is a programming language designed for the Automata Theory and Formal Languages classroom. Among its types are regular expressions, finite-state machine (e.g., finite-state automata, pushdown automata, and Turing machine), and grammars (e.g., regular grammar, context-free grammar, and context-sensitive grammar). It also defines constants like `EMP` that denotes the empty word (i.e., a word of length 0). In contrast, a nonempty word is a `(listof symbol)`. Each symbol in a nonempty word is a member of an alphabet, Σ, that may contain symbols representing lowercase letters in the Roman alphabet, numbers, or special characters like `&`, `!`, `$`, and `*`.

An important feature of `FSM` is that nondeterminism is a feature. That is, programmers are not burdened with implementing nondeterminism. Instead, nondeterminism is built into language primitives.

3.1 Regular Expressions

A regular expression, over an alphabet Σ, is an `FSM` type defined as follows[1]:

1. (empty-regexp)
2. (singleton-regexp "a"), where a∈Σ
3. (union-regexp r1 r2), where r1 and r2 are regular expressions
4. (concat-regexp r1 r2), where r1 and r2 are regular expressions
5. (kleenestar-regexp r), where r is a regular expression

Each regular expression subtype is built using a distinct constructor. The language of a regular expression, `r`, is denoted by `L(r)`. It contains all the words that can be generated with `r`. A language that is described by a regular expression is called a *regular language*.

The first regular expression describes the following language:

[1]The regular expression for the empty language, `(null-regexp)`, is not introduced until needed later in the course when transforming a nondeterministic finite-state automaton into a regular expression. In this manner, discussions about how operations are performed with `(null-regexp)` (e.g., `(concat A (null-regexp))`) are postponed until students are more familiar with regular expressions.

$$L((\text{empty-regexp})) = \{\text{EMP}\} = \{\varepsilon\}$$

That is, it is a language that only contains the empty word.

The constructor `singleton-regexp` is used to build a regular expression for any element in Σ. It takes as input a string representing an element in Σ. A singleton regular expression describes the following language:

$$L((\text{singleton-regexp ”a”})) = \{a\}$$

That is, it is the language that only contains a.

If `r1` and `r2` are regular expressions then a regular expression for `L(r1)` ∪ `L(r2)` is built using `union-regexp`. It describes the following language:

$$L((\text{union-regexp r1 r2})) = \{w \mid w{\in}L(r1) \vee w{\in}L(r2)\}$$

That is, it represents the language that contains all the words in `L(r1)` and all the words in `L(r2)`.

If `r1` and `r2` are regular expressions then `concat-regexp` builds a regular expression for `L(r1)L(r2)`. It describes the following language:

$$L((\text{concat-regexp r1 r2})) = \{w_1w_2 \mid w_1{\in}L(r1) \wedge w_2{\in}L(r2)\}$$

That is, it is the language that contains all words constructed by concatenating a word in `L(r1)` and a word in `L(r2)`.

If `r` is a regular expression then a regular expression for zero or more concatenations of words in `L(r)` is built using `kleenestar-regexp`. It represents the following language:

$$L((\text{kleenestar-regexp r})) = \{\{\text{EMP}\} \cup \{w_1w_2 \ldots w_n \mid w_1,w_2,\ldots,w_n \in L(r) \wedge n \geq 1\}\}$$

That is, it is the language that contains all words constructed by concatenating zero or more words in `L(r)`.

Students are made aware that `FSM` provides informative error messages to help overcome the misuse of regular expression constructors[11]. When a constructor is misused an error is thrown. This is a sampling of `FSM` error messages for misuse of regular expression constructors:

```
> (union-regexp 2 (singleton-regexp 'w))
the input to the regexp #(struct:singleton-regexp w) must be a string
> (union-regexp (empty-regexp) 3)
3 must be a regexp to be a valid second input to
union-regexp #(struct:empty-regexp) 3
> (concat-regexp 3 (empty-regexp))
3 must be a regexp to be a valid first input to
concat-regexp 3 #(struct:empty-regexp)
> (kleenestar-regexp "A U B")
"A U B" must be a regexp to be a valid input to kleenestar-regexp
> (singleton-regexp 1)
the input to the regexp #(struct:singleton-regexp 1) must be a string
```

Observe that the error messages are not prescriptive. This is important because it is impossible to discern the intention of the programmer [6, 7]. Proposing a solution to the bug may lead the programmer down the wrong path to solve it in a manner consistent with her design.

`FSM` does not burden programmers with implementing nondeterminism for regular expressions. Nondeterminism in `FSM` regular expressions is a language feature that students do not need to know how to implement (much like they do not need to know how to implement features in their favorite programming language). To this end, the following primitives are provided:

(pick-regexp r): Nondeterministically returns a nested sub-`regexp` from the given `union-regexp`. This includes any nested `union-regexps` in a chain of `union-regexps`. For example, if the `union-regexp` is `(union-regexp r1 (union-regexp r2 (union-regexp r3 r4)))` then the selected `regexp` may be any of `r1`–`r4`.

(pick-reps n): Nondeterministically generate a natural number in [0..n].

(gen-regexp-word r): Nondeterministically generates a word in the language of the given `regexp`.

For convenience, `FSM` also provides the following function to generate a word from a `singleton-regexp`:

(convert-singleton r): Converts the given `singleton-regexp` to a word of length 1.

Finally, `FSM` addresses the printed representation of regular expressions. By "printable" we mean a string fit for humans to read. The following table outlines the printable forms of regular expressions:

Regular Expression	Printable Form
`(empty-regexp)`	`"ε"`
`(singleton-regexp a)`	`"a"`
`(union-regexp r1 r2)`	(string-append (printable-regexp r1) `"U"` (printable-regexp r2))
`(concat-regexp r1 r2)`	(string-append (printable-regexp r1) (printable-regexp r2))
`(kleenestar-regexp r)`	(string-append (printable-regexp r1) `"*"`)

The `FSM` function `printable-regexp` returns a string representing the regular expression it is given as input. The following interactions illustrate how `printable-regexp` works:

```
> (printable-regexp (empty-regexp))
"ε"
> (printable-regexp (singleton-regexp "z"))
"z"
> (printable-regexp (union-regexp
                      (singleton-regexp "z")
                      (union-regexp (singleton-regexp "1")
                                    (singleton-regexp "q"))))
"(z U (1 U q))"
> (printable-regexp (concat-regexp (singleton-regexp "i")
                                   (singleton-regexp "i")))
"ii"
> (printable-regexp (kleenestar-regexp
                      (concat-regexp (singleton-regexp "a")
                                     (singleton-regexp "b"))))

"(ab)*"
```

3.2 Regular Expression Selectors and Predicates

The `FSM` selector functions for regular expressions are:

singleton-regexp-a: Extracts the embedded string

kleenestar-regexp-r1: Extracts the embedded regular expression
union-regexp-r1: Extracts the first embedded regular expression
union-regexp-r2: Extracts the second embedded regular expression
concat-regexp-r1: Extracts the first embedded regular expression
concat-regexp-r2: Extracts the second embedded regular expression

The following predicates are defined to distinguish among the regular expression subtypes:

```
empty-regexp?     singleton-regexp?     kleenestar-regexp?
union-regexp?     concat-regexp?
```

Each consumes one value of any type and returns a Boolean. They return true if the input is a regular expression of the subtype tested. Otherwise, they return false.

Armed with the constructors, selectors, and predicates for regular expressions we can write a template for functions on a regular expression:

```
;; regexp ... → ...
;; Purpose: ...
(define (f-on-regexp rexp ...)
  (cond [(empty-regexp? rexp) ...]
        [(singleton-regexp? rexp) ...(singleton-regexp-a rexp)...]
        [(kleenestar-regexp? rexp)
         ...(f-on-regexp (kleenestar-regexp-r1 rexp))...]
        [(union-regexp? rexp)
         ...(f-on-regexp (union-regexp-r1 rexp))...
         ...(f-on-regexp (union-regexp-r2 rexp))...]
        [else ...(f-on-regexp (concat-regexp-r1 rexp))...
              ...(f-on-regexp (concat-regexp-r2 rexp))...]))
```

The function template reflects the structure of regular expressions and suggests using structural recursion to process a regular expression.

4 Programming with Regular Expressions

Given that `FSM` has a regular expression type, their design may follow a top-down or a bottom-up divide-and-conquer approach. The idea is to define a regular expression by parts. Just like a program is composed of one or more functions, a regular expression is composed of one or more regular expressions. In this section we explore how to design and implement regular expressions.

4.1 All Words Ending with an `a`

The first (gentle) exercise building a finite representation for an infinite language starts with the following language over Σ = {`a b`}:

$$L = \{w \mid w \text{ ends with an a}\}$$

Students are asked if a regular expression for `L` can be programmed. To start, a top-down design is followed. Every word in `L` must have at least one `a` at the end. Before the last `a` there can be an arbitrary number of `a`s and `b`s. Assuming a regular expression can be developed for both parts, the regular expression for `L` that concatenates the languages for each part may be written as follows:

Figure 1 The FSM program for L = {w | w ends with an a}.

```
(define A (singleton-regexp "a"))

(define B (singleton-regexp "b"))

(define AUB (union-regexp A B))

(define AUB* (kleenestar-regexp AUB))

(define ENDS-WITH-A  (concat-regexp AUB* A))

(check-equal? (printable-regexp ENDS-WITH-A) "(a U b)*a")
```

```
(define ENDS-WITH-A  (concat-regexp AUB* A))
```

The regular expression `A` must represent `a`. This is defined as follows:

```
(define A (singleton-regexp "a"))
```

`AUB*` must represent an arbitrary number of elements. This suggests defining a `kleenestar-regexp`. This may be done as follows:

```
(define AUB* (kleenestar-regexp AUB))
```

`AUB` represents a choice between a word in `L(A)` and a word in `L(B)`. Having a choice suggests defining a `union-regexp`:

```
(define AUB (union-regexp A B))
```

Finally, `B` represents the singleton regular expression for `"b"`. This is done as follows:

```
(define B (singleton-regexp "b"))
```

The complete program for a regular expression for `L` is displayed in Figure 1. The unit test is written using `check-equal?` provided by, `rackunit`, `Racket`'s unit-testing framework.

4.2 Binary Numbers

Next students are asked to consider a slightly more complex language that ought to be, at least informally, familiar to them. They consider the following language:

BIN-NUMS = {w | w is a binary number without leading zeroes}

Although the above definition may sound clear to students, it is lacking. It does not provide any details about the structure of the binary numbers in the language nor any indication on how to build such numbers. Class turns to formally define BIN-NUMS using a regular expressions. To provide a different development perspective, a bottom-up divide-and-conquer approach is used.

4.2.1 Implementing a Regular Expression

Based on the problem statement the following observations are made:

Figure 2 A program defining binary numbers without leading zeroes

```
(define ZERO (singleton-regexp "0"))

(define ONE  (singleton-regexp "1"))

(define 0U1* (kleenestar-regexp (union-regexp ZERO ONE)))

(define STARTS1 (concat-regexp ONE 0U1*))

(define BIN-NUMS (union-regexp ZERO STARTS1))

(check-equal? (printable-regexp BIN-NUMS) "(0 U 1(0 U 1)*)")
```

1. Σ = {0 1}
2. The minimum length of a binary number is 1
3. A binary number with a length greater than 1 cannot start with 0

The second observation informs us that the empty regular expression is not part of the language. The third observation informs us that there are two subtypes of binary numbers in the set: `0` and those starting with 1 followed by an arbitrary number of `0`s and `1`s.

The simplest regular expressions needed are for the elements of Σ. These are all singleton regular expressions:

```
(define ZERO (singleton-regexp "0"))   (define ONE  (singleton-regexp "1"))
```

An arbitrary number of `0`s and `1`s may be represented using a union and a Kleene star regular expression:

```
(define 0U1* (kleenestar-regexp (union-regexp ZERO ONE)))
```

With the above definition, a regular expression for 1 followed by an arbitrary number of `0`s and `1`s is:

```
(define STARTS1 (concat-regexp ONE 0U1*))
```

Finally, `BIN-NUMS` may be implemented by a union regular expression to provide a choice among the subtypes. Figure 2 displays the complete program to define `BIN-NUMS`.

4.2.2 Generating `BIN-NUMS` Words

Students are asked to compare the two formulations for `BIN-NUMS` and determine which is more useful:

- BIN-NUMS = {w | w is a binary number without leading zeroes}
- BIN-NUMS = $(0 \cup 1(0 \cup 1)^*)$

The truth is that both are useful. The first provides a quick intuitive understanding of what the language represents. It lacks, however, any description for constructing words. In this regard, the second formulation is more useful. It describes an algorithm for constructing binary numbers without leading zeroes. Either generate 0 or generate 1 followed by an arbitrary number of `0`s or `1`s.

This means that a function to generate a random word in `BIN-NUMS` can and ought to be implemented. We shall follow the steps of the design recipe for a function to write this program [1, 9]. To simplify the discussion the default number of Kleene star repetitions shall be 10. Based on this design idea, the next steps of the design recipe are outlined as follows:

```
;; [natnum>0] → BIN-NUMS
;; Purpose: Generate a binary number without leading zeroes
(define (generate-bn . n)

  (define MAX-KS-REPS (if (null? n) 10 (first n)))

  ;; regexp → word
  ;; Purpose: Generate a word representing a valid binary number,
  ;;          such that the number of Kleene star repetitions is
  ;;          in [0..MAX-KS-REPS].
  (define (gen-word r) ...)

 (gen-word BIN-NUMS))
```

The signature and function header define, respectively, a single optional argument for the maximum number of repetitions for a Kleene star regular expression. The purpose statement clearly and briefly describes the problem solved. A local constant is defined for the Kleene star repetitions with 10 as the default value. Finally, the local function `gen-word` generates a binary number and is designed by specializing the template for functions on a regular expression.

The next step of the design recipe is to write unit tests. We are unable, however, to write tests using `check-equals?` because the program randomly generates elements in `BIN-NUMS`. To test such a function we use property-based testing. That is, we shall test that the generated words have the expected properties. To write tests we use `rackunit`'s `check-pred`. `check-pred` requires a predicate to test and input for the predicate. If the predicate holds the test passes. If the predicate does not hold the test fails and a failed test report is generated. Any word, `w`, generated by `generate-bn` must have the following properties:

1. `w` is a list (i.e., it cannot be `EMP`)
2. $1 \leq$ `(length w)`
3. `w` is `'(0)` or `(first w)` is 1
4. `w` only contains `0`s and `1`s

To test that a generated word represents a binary number we ignore the length limit. Following the steps of the design recipe yields the following predicate:

```
;; word → Boolean
;; Purpose: Test if the given word is in L(BIN-NUMS)
(define (is-bin-nums? w)
  (and (list? w)
       (<= 1 (length w))
       (or (equal? w '(0)) (= (first w) 1))
       (andmap (λ (bit) (or (= bit 0) (= bit 1))) w)))

(check-equal? (is-bin-nums? '()) #f)
(check-equal? (is-bin-nums? '(0 0 0 1 1 0 1 0)) #f)
(check-equal? (is-bin-nums? '(0)) #t)
(check-equal? (is-bin-nums? '(1 0 0 1 0 1 1)) #t)
(check-equal? (is-bin-nums? '(1 1 1 0 1 0 0 0 1 1 0 1)) #t)
```

Figure 3 The function to generate words in BIN-NUMS.

```
#lang fsm
;; [natnum>0] → word
;; Purpose: Generate a binary number without leading zeroes
;;          unless its 0
(define (generate-bn . n)
  (define MAX-KS-REPS (if (null? n) 10 (first n)))

  ;; regexp → word
  ;; Purpose: Generate a word representing a valid binary number,
  ;;          such that the number of Kleene star repetitions is
  ;;          in [0..MAX-KS-REPS]
  (define (gen-word r)
    (cond [(singleton-regexp? r) (convert-singleton r)]
          [(concat-regexp? r)
           (let [(w1 (gen-word (concat-regexp-r1 r)))
                 (w2 (gen-word (concat-regexp-r2 r)))]
             (append w1 w2))]
          [(union-regexp? r) (gen-word (pick-regexp r))]
          [(kleenestar-regexp? r)
           (flatten (build-list
                     (pick-reps MAX-KS-REPS)
                     (λ (i) (gen-word (kleenestar-regexp-r1 r)))))]))
   (gen-word BIN-NUMS))
```

This predicate is used to write the tests for generate-bn. Anything returned by generate-bn must satisfy is-bin-nums?. The tests are:

```
(check-pred is-bin-nums? (generate-bn))
(check-pred is-bin-nums? (generate-bn))
(check-pred is-bin-nums? (generate-bn))
(check-pred is-bin-nums? (generate-bn))
(check-pred is-bin-nums? (generate-bn))
```

Although the tests all look the same they are not the same test. Recall that (generate-bn) is nondeterministic (i.e., the output value cannot be predicted). Therefore, each test above is for a different word (not necessarily distinct) returned by (generate-bn).

The function gen-word must be able to process any regexp subtype that is part of the BIN-NUMS: singleton-regexp, concat-regexp, union-regexp, and kleenestar-regexp. The function template to process a regexp is specialized. To process a singleton-regexp, convert-singleton is used. To process a concat-regexp, two words are generated, using each embedded regexp, and appended. To process a union-regexp, a recursive call is made with one of the sub-regexps nondeterministically chosen by pick-regexp. To process a kleenestar-regexp, the number of binary numbers to generate is nondeterministically chosen using pick-reps. A list containing that number of binary numbers is generated and flattened. The result of this design is displayed in Figure 3.

5 Generating Words in the Language Defined by a Regular Expression

The development of generate-bn confirms that a regular expression, r, describes a construction algorithm for words in L(r). This suggests that the algorithm can be generalized to generate an arbitrary word in the language of an arbitrary regular expression.

5.1 Design Idea

The function takes as input a regular expression and an optional natural number for the maximum number of Kleene star repetitions. It returns a word. A constant, MAX-KLEENESTAR-REPS, is locally defined for the maximum number of Kleene star repetitions. If the optional natural number is not provided the default value of the constant is arbitrarily defined to be 20. Building on the experience with generating binary numbers, there is a local function, generate, to generate a word.

As suggested by the function template to process a regexp, generate must distinguish among the regular expression subtypes to generate the word. If the input is the empty regular expression then the only word that may be generated is, EMP, the empty word. If given a singleton regular expression then convert-singleton is used to generate a word of length 1 from the embedded string.

To process a Kleene star regular expression, a list of words is generated using the embedded regexp. The length of the list is nondeterministically chosen using pick-reps. Once generated, the list of words is filtered for empty words and flattened. If the resulting list is empty then EMP is returned. Otherwise, the resulting list is returned.

To process a union regular expression, pick-regexp is used to nondeterministically select one of the expressions in the union. A recursive call is made with the selected regular expression to generate the word.

To process a concat-regexp, a word is generated using each of the embedded regular expressions. If both generated words are EMP then EMP is returned. If either word is EMP then the other word is returned. Otherwise, the two generated words are appended and returned.

5.2 Signature, Purpose, and Function Header

Students are reminded that the signature, purpose statement, and function header collectively provide documentation that explains to any reader of the code what the function is expected to do. The next steps of the design recipe are satisfied as follows:

```
;; regexp [natnum] → word
;; Purpose: Generate a random word in the language
;;          of the given regexp such that the number
;;          of repetitions generated from a Kleene
;;          star regular expression does not exceed
;;          the given optional number or, otherwise,
;;          20.
(define (gen-word rexp . reps)
```

5.3 Tests

To simplify the development of tests we use ENDS-WITH-A from Figure 1 and BIN-NUMS from Figure 2. Given that we are designing a nondeterministic function, property-based testing is employed. This means

we must design and implement a predicate for words ending with an a just like is-bin-nums? is designed for BIN-NUMS. If the given word is a list, has a length greater than or equal to 1, and its last element is an a then it is a word in L(ENDS-WITH-A). Following the steps of the design recipe yields this predicate:

```
;; word → Boolean
;; Purpose: Test if the given word is in ENDS-WITH-A
(define (is-ends-with-a? w)
  (and (list? w) (>= (length w) 1) (eq? (last w) 'a)))

(check-equal? (is-ends-with-a? '(a)) #t)
(check-equal? (is-ends-with-a? '(b b a)) #t)
(check-equal? (is-ends-with-a? '(a b b a b a)) #t)
(check-equal? (is-ends-with-a? '()) #f)
(check-equal? (is-ends-with-a? '(b b b)) #f)
(check-equal? (is-ends-with-a? '(a a a a b)) #f)
```

The tests for gen-word are:

```
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS))
(check-pred is-bin-nums? (gen-word BIN-NUMS 30))
(check-pred is-bin-nums? (gen-word BIN-NUMS 50))

(check-pred is-ends-with-a? (gen-word ENDS-WITH-A))
(check-pred is-ends-with-a? (gen-word ENDS-WITH-A))
(check-pred is-ends-with-a? (gen-word ENDS-WITH-A 18))
(check-pred is-ends-with-a? (gen-word ENDS-WITH-A 7))
```

5.4 Function Body

The next step of the design requires writing the function's body. This is done by specializing the cond-expression in the template for functions on a regular expression. We independently present the design of each stanza.

For the empty regular expression the only word that can be generated is EMP. The corresponding stanza is:

```
[(empty-regexp? rexp) EMP]
```

For a singleton-regexp, a word is generated using convert-singleton:

```
[(singleton-regexp? rexp) (convert-singleton rexp)]
```

For a kleenestar-regexp, the length of the word is nondeterministically chosen using pick-reps. A list of words of the chosen length, generated using the embedded regular expression, is filtered to remove all EMPs and flattened. If the flattened list is empty then EMP is returned. Otherwise, the flattened list is returned. The required code is:

```
[(kleenestar-regexp? rexp)
 (let*
```

```
[(reps (pick-reps MAX-KLEENESTAR-REPS))
 (low (flatten
       (filter
        (λ (w) (not (eq? w EMP)))
         (build-list
          reps
          (lambda (i)
           (gen-word (kleenestar-regexp-r1 rexp)))))))]
(if (empty? low) EMP low))]
```

For a union regular expression, a word is generated by nondeterministically picking a regular expression from the options in the union and making a recursive call with the maximum number of repetitions:

```
[(union-regexp? rexp) (gen-word (pick-regexp rexp))]
```

For a concatenation regular expression, two words are generated using each embedded regular expression. The words are examined as described in the design idea to return the generated word. The default stanza of the conditional is:

```
[else
 (let [(w1 (gen-word (concat-regexp-r1 rexp)))
       (w2 (gen-word (concat-regexp-r2 rexp)))]
   (cond [(and (eq? w1 EMP) (eq? w2 EMP)) EMP]
         [(eq? w1 EMP) w2]
         [(eq? w2 EMP) w1]
         [else (append w1 w2)]))]
```

It is suggested to students to take time to appreciate what has been achieved. A regular expression, simultaneously, is a description of, L, a regular language and a description of a construction algorithm for the words in L. It is suggested to students that a construction algorithm is an elegant way of specifying a language.

6 Regular Expression Applications

Before ending the module on regular expressions, students are introduced to one or more applications. This is something that always helps students appreciate the material they are learning. At this point, students realize that regular expressions capture a pattern for the construction of languages. As such, they are told, that regular expressions are easily found in many areas of Computer Science and, indeed, in life. It is important to note that the term regular expression is used differently in different domains. That is, a regular expression is not always defined as defined in an automata theory course. Generally, all definitions have union, concatenation, and Kleene star operations. They, however, also include other operations. These other operations may provide the ability to describe languages that are not regular. The syntax, of course, also varies. Consider, for example, the following `Perl` code snippet:

```
$foo =~ m/fsm/
```

This expression evaluates to true if `$foo` contains `fsm`. Put differently, it evaluates to true if `$foo`'s value is a word in the following language:

$$L = \{w \mid x', y' \in \Sigma^* \land w = x'\texttt{fsm}y'\}$$

Clearly, `L` is a regular language. Regular expressions in `Perl`, however, are strong enough to match languages that are not regular. Therefore, students are advised that when speaking about regular expressions it is important to be precise.

Regular expressions may be used to describe, for example, internet addresses, proteins, decimal numbers, and patterns to search for in text among others. To illustrate the use of regular expressions we explore the problem of generating passwords. As always, we follow the steps of the design recipe to write a password generating function.

6.1 Data Definitions

A password is a string that:

- Has length ≥ 10
- Includes at least one of each: lowercase letter, uppercase letter, and special character (i.e., $, &, !, or *)

Based on this definition, the sets for lowercase letters, uppercase letters, and special characters are defined as follows:

```
(define lowers '(a b c d e f g h i j k l m n o p q r s t u v w x y z))
(define uppers '(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z))
(define spcls  '($ & ! *))
```

The corresponding sets of regular expressions are defined as:

```
(define lc  (map (λ (lcl) (singleton-regexp (symbol->string lcl))) lowers))
(define uc  (map (λ (ucl) (singleton-regexp (symbol->string ucl))) uppers))
(define spc (map (λ (sc) (singleton-regexp (symbol->string sc))) spcls))
```

How is a password defined? To create passwords we need a regular expression. Once a password is generated it can be transformed into a string. The order in which lowercase letters, uppercase letters, and special characters appear is arbitrary. There must be, however, an, `L`, lowercase letter, a, `U`, uppercase letter, and a, `S`, special character. There are six different orderings these required elements may appear in:

```
L U S    U L S    S U L    L S U    U S L    S L U
```

Before and after each required element there may be an arbitrary number lowercase letters, uppercase letters, and special characters. A union regular expression is needed for the lowercase letters, for the uppercase letters, for the special characters, and for the arbitrary characters that may appear between required characters. It is emphasized to students that a union regular expression is used because it provides the ability to choose any element. These may be defined as follow:

```
(define LOWER (create-union-regexp lc))
(define UPPER (create-union-regexp uc))
(define SPCHS (create-union-regexp spc))
(define ARBTRY (kleenestar-regexp
                (union-regexp LOWER (union-regexp UPPER SPCHS))))
```

The creation of a chain of union regular expressions is delegated to, `create-union-regexp`, an auxiliary function (to be designed and implemented). It is now possible to define a regular expression for each of the six orderings of required elements. For instance, the regular expression for words that have the required lowercase letter first, the uppercase letter second, and the special character third is defined as follows:

```
(define LUS (concat-regexp
              ARBTRY
              (concat-regexp
                LOWER
                (concat-regexp
                  ARBTRY
                  (concat-regexp
                    UPPER
                    (concat-regexp ARBTRY (concat-regexp SPCHS ARBTRY)))))))
```

The regular expressions for the remaining 5 orderings are similarly defined.

The language of passwords is a word in any of the languages for the different orderings of required elements. It is defined using a union regular expression:

```
(define PASSWD (union-regexp
                 LUS
                 (union-regexp
                   LSU
                   (union-regexp
                     SLU (union-regexp SUL (union-regexp USL ULS))))))
```

Finally, in order to prevent generated passwords from getting unwieldy long `MAX-KLEENESTAR-REPS` is redefined as follows:

```
(define MAX-KLEENESTAR-REPS 5)
```

6.2 Design Idea

The constructor for a password takes no input and returns a string. A potential new password is locally defined. A word is generated by applying `gen-regexp-word` (developed in Section 5) to `PASSWD` and then converting the result to a string. If the length of the string is greater than or equal to 10 then it is returned as the generated password. Otherwise, a new password is generated.

6.3 Function Definition

Following the steps of the design recipe yields the following function definition:

```
;;  → string
;; Purpose: Generate a valid password
(define (generate-password)
  (let [(new-passwd (passwd->string (gen-regexp-word PASSWD)))]
    (if (>= (string-length new-passwd) 10)
        new-passwd
        (generate-password))))
```

6.4 Tests

Given that the function is nondeterministic, property-based testing is used. For this, a predicate that takes as input a string representing a password is needed. The given password is converted into a list of symbols representing the characters in the string. This list must have a length of at least 10 and contain

one element in each of the following: `lowers`, `uppers`, and `spcls`. Following the steps of the design recipe yields:

```
;; string → Boolean
;; Purpose: Test if the given string is a valid password
(define (is-passwd? p)
  (let [(los (str->los p))]
    (and (>= (length los) 10)
         (ormap (λ (c) (member c los)) lowers)
         (ormap (λ (c) (member c los)) uppers)
         (ormap (λ (c) (member c los)) spcls))))
```

Converting the password to a list is delegated to, `str->los`, an auxiliary function.

Sample tests using `check-pred` are:

```
(check-pred is-passwd? (generate-password))
(check-pred is-passwd? (generate-password))
(check-pred is-passwd? (generate-password))
(check-pred is-passwd? (generate-password))
(check-pred is-passwd? (generate-password))
```

6.5 Auxiliary Functions

Three auxiliary functions are needed: `create-union-regexp`, `str->los`, and `passwd->string`. The function `create-union-regexp` takes as input a list of regular expressions and returns a union regular expression. If the length of the given list is less than 2 an error is thrown because at least two regular expressions are needed for the union of regular expressions. If the given list only has two elements then a union regular expression is constructed with the two regular expressions in the list. If the given list has a length greater than to 2 then a union regular expression is constructed with the first regular expression in the list and the union regular expression obtained from recursively processing the rest of the list. Following the steps of the design recipe yields:

```
;; (listof regexp) → union-regexp throws error
;; Purpose: Create union-regexp using given list of regular expressions
(define (create-union-regexp L)
  (cond [(< (length L) 2)
         (error "create-union-regexp: list too short")]
        [(empty? (rest (rest L))) (union-regexp (first L) (second L))]
        [else (union-regexp (first L) (create-union-regexp (rest L)))]))

;; Tests
(check-equal? (create-union-regexp (list (first lc) (first uc)))
              (union-regexp (singleton-regexp "a") (singleton-regexp "A")))
(check-equal? (create-union-regexp (list (first lc) (fourth uc) (third spc)))
              (union-regexp
               (singleton-regexp "a")
               (union-regexp (singleton-regexp "D") (singleton-regexp "!"))))
```

The function `str->los` takes as input a string and returns a list of symbols. The given string is converted to a list (of characters). Each character in the resulting list is converted to a symbol using `map`.

The function given to map consumes a character and first converts the character into a string and then converts the string into a symbol. Following the steps of the design recipe yields:

```
;; string → (listof symbol)
;; Purpose: Convert the given string to a list of symbols
(define (str->los str)
  (map (λ (c) (string->symbol (string c))) (string->list str)))

;; Tests
(check-equal? (str->los "") '())
(check-equal? (str->los "a!Cop") '(a ! C o p))
```

Finally, the function passwd->string converts a given word representing a password into a string. First, the given word is converted into a list of characters using map. The function given to map converts a symbol into a character by transforming the symbol into a string, transforming the string into a list of characters, and finally taking the first (and only) element in the list of characters. Second, the list of characters produced by map is converted into a string. The steps of the design recipe produce:

```
;; word → string
;; Purpose: Convert the given password to a string
(define (passwd->string passwd)
  (list->string
   (map (λ (s) (first (string->list (symbol->string s)))) passwd)))

;;Tests
(check-equal? (passwd->string '(a j h B ! ! y y t c)) "ajhB!!yytc")
(check-equal? (passwd->string '($ u t q x ! J i n * K C)) "$utqx!Jin*KC")
```

6.6 Running the Tests

The students run the program and confirm that all the tests pass. In addition, students are encouraged to generate a few passwords. These are sample passwords generated:

```
> (generate-password)
"&&!$m*F!&$*"
> (generate-password)
"!e*e!*oS!lq$"
> (generate-password)
"!y*$r!C&*d$"
> (generate-password)
"&&!p$rUA$*"
> (generate-password)
"W&*!eKY**D"
> (generate-password)
"vxY*We!Wx*&&u"
```

Students feel a sense of accomplishment seeing the results. The feeling is that passwords generated are robust and it is unlikely that anyone would be able to guess any of them.

7 Concluding Remarks

The work presented outlines a didactic approach for introducing students to regular expressions. Unlike most Formal Languages and Automata Theory textbooks that emphasize mathematical properties to simplify regular expressions, the work presented emphasizes algorithm design and implementation. This helps keep Computer Science students motivated and engaged. Students can program regular expressions and algorithms based on regular expressions in `FSM`. The importance of presenting practical and relevant applications is not ignored. Nearly all students appreciate, for example, the development of the program to generate passwords. Most students comment that it is an approach they had never had thought about before and are eager for other examples. To address this, homework exercises involving token generation or DNA pattern generation are effective.

Future work will address creating a database of examples instructors and students may draw upon for practice or presentation. The goal is to have a diverse set of examples–after all, for instance, not every student may find password generation exciting. In addition, extensions to `FSM` are being considered. For example, it may be useful for some students to endow `FSM` with a primitive to generate words in the language of a given regular expression.

References

[1] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2018): *How to Design Programs: An Introduction to Programming and Computing*, Second edition. MIT Press, Cambridge, MA, USA.

[2] Matthew Flatt, Robert Bruce Findler & PLT: *The Racket Guide*. Available at `https://docs.racket-lang.org/guide/`. Last accessed 2023-07-07.

[3] John E. Hopcroft & Jeffrey D. Ullman (1979): *Introduction to automata theory, languages and computation*. Addison-Wesley.

[4] Harry R. Lewis & Christos H. Papadimitriou (1997): *Elements of the Theory of Computation*, 2nd edition. Prentice Hall PTR, Upper Saddle River, NJ, USA, doi:10.1145/300307.1040360.

[5] Peter Linz (2011): *An Introduction to Formal Languages and Automata*, 5th edition. Jones and Bartlett Publishers, Inc., USA.

[6] Guillaume Marceau, Kathi Fisler & Shriram Krishnamurthi (2011): *Measuring the Effectiveness of Error Messages Designed for Novice Programmers*. In: *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, ACM, New York, NY, USA, pp. 499–504, doi:10.1145/1953163.1953308.

[7] Guillaume Marceau, Kathi Fisler & Shriram Krishnamurthi (2011): *Mind Your Language: On Novices' Interactions with Error Messages*. In: *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, ACM, New York, NY, USA, pp. 3–18, doi:10.1145/2048237.2048241.

[8] John C. Martin (2003): *Introduction to Languages and the Theory of Computation*, 3 edition. McGraw-Hill, Inc., New York, NY, USA.

[9] Marco T. Morazán (2022): *Animated Problem Solving - An Introduction to Program Design Using Video Game Development*. Texts in Computer Science, Springer, doi:10.1007/978-3-030-85091-3.

[10] Marco T. Morazán & Rosario Antunez (2014): *Functional Automata - Formal Languages for Computer Science Students*. In James Caldwell, Philip K. F. Hölzenspies & Peter Achten, editors: *Proceedings 3rd International Workshop on Trends in Functional Programming in Education*, *EPTCS* 170, pp. 19–32, doi:10.4204/EPTCS.170.2.

[11] Marco T. Morazán & Josephine A. Des Rosiers (2019): *FSM Error Messages*. *EPTCS* 295, pp. 1–16, doi:10.4204/EPTCS.295.1. Available at `https://arxiv.org/abs/1906.11421v1`.

[12] Elaine Rich (2019): *Automata, Computability and Complexity: Theory and Applications*. Pearson Prentice Hall.

[13] Michael Sipser (2013): *Introduction to the Theory of Computation*, 3rd edition. Cengage Learning.

DISCO: A Functional Programming Language for Discrete Mathematics

Brent A. Yorgey
Hendrix College
Conway, Arkansas, USA
yorgey@hendrix.edu

DISCO is a pure, strict, statically typed functional programming language designed to be used in the setting of a discrete mathematics course. The goals of the language are to introduce students to functional programming concepts early, and to enhance their learning of mathematics by providing a computational platform for them to play with. It features mathematically-inspired notation, property-based testing, equirecursive algebraic types, subtyping, built-in list, bag, and finite set types, a REPL, and student-focused documentation. DISCO is implemented in Haskell, with source code available on GitHub,[1] and interactive web-based REPL available through replit.[2]

1 Introduction

Many computer science curricula at the university level include a *discrete mathematics* course as a core requirement [CM13]. Often taken in the first or second year, a discrete mathematics course introduces mathematical structures and techniques of foundational importance in computer science, such as induction and recursion, set theory, logic, modular arithmetic, functions, relations, and graphs. In addition, it sometimes serves as an introduction to writing formal proofs. Although there is wide agreement that discrete mathematics is foundational, students often struggle to see its relevance to computer science.

Functional programming is a style of programming, embodied in languages such as Haskell, Standard ML, OCaml, Scala, F#, and Racket, which emphasizes functions (*i.e.* input-output processes) rather than sequences of instructions. It enables working at high levels of abstraction as well as rapid prototyping and refactoring, and provides a concise and powerful vocabulary to talk about many other topics in computer science. It is becoming critical to expose undergraduate students to functional programming early, but many computer science programs struggle to make space for it. The Association for Computing Machinery's 2013 curricular guidelines [CM13] do not even include functional programming as a core topic.

One creative idea is to combine functional programming and discrete mathematics into a single course. This is not a new idea [Wai92, Hen02, SW02, DE04, OHP06, Van11, Van13, Van17, Xin08], and even shows up in the 2007 model curriculum of the Liberal Arts Computer Science Consortium [Lib07]. The benefits of such an approach are numerous:

- It allows functional programming to be introduced at an early point in undergraduates' careers, since discrete mathematics is typically taken in the first or second year. This allows ideas from functional programming to inform students' thinking about the rest of the curriculum. By contrast, when functional programming is left until later in the course of study, it is in danger of being seen as esoteric or as a mere curiosity.

[1] https://github.com/disco-lang/disco
[2] https://replit.com/@BrentYorgey/Disco#README.md

Elena Machkasova (Ed.): International Workshop on Trends in Functional Programming in Education 2023 (TFPIE 2023)
EPTCS 382, 2023, pp. 64–81, doi:10.4204/EPTCS.382.4

© B. A. Yorgey
This work is licensed under the Creative Commons Attribution License.

- The two subjects complement each other well: discrete math topics make good functional programming exercises, and ideas from functional programming help illuminate topics in discrete mathematics.

- In a discrete mathematics course with both mathematics and computer science majors, mathematics majors can have a "home turf advantage" since the course deals with topics that may be already familiar to them (such as writing proofs), whereas computer science majors may struggle to connect the course content to computer science skills and concepts they already know. Including functional programming levels the playing field, giving both groups of students a way to connect the course content to their previous experience. Computer science majors will be more comfortable learning mathematical concepts that they can play with computationally; mathematics majors can leverage their experience with mathematics to learn a bit about programming.

- It is just plain fun: using programming enables interactive exploration of mathematical concepts, which leads to higher engagement and increased retention.

However, despite its benefits, this model is not widespread in practice. This may be due partly to lack of awareness, but there are also some real roadblocks to adoption that make it impractical or impossible for many departments.

- Existing functional languages—such as Haskell, Racket, OCaml, or SML—are general-purpose languages which (with the notable exception of Racket) were not designed specifically with teaching in mind. The majority of their features are not needed in the setting of discrete mathematics, and teachers must waste a lot of time and energy explaining incidental detail or trying to hide it from students.

- Again with the notable exception of Racket, tooling for existing functional languages is designed for professional programmers, not for students. The systems can be difficult to set up, generate confusing error messages, and are generally designed to facilitate efficient production of code rather than interactive exploration and learning.

- As with any subject, effective teaching of a functional language requires expertise in the language and its use, or at least thorough familiarity, on the part of the instructor. General-purpose functional languages are large, complex systems, requiring deep study and years of experience to master. Even if only a small part of the language is presented to students, a high level of expertise is still required to be able to select and present a relevant subset of the language and to help students navigate around the features they do not need. For many instructors, spending years learning a general-purpose functional language just to teach discrete mathematics is a non-starter. This is especially a problem at institutions where the discrete mathematics course is taught by mathematics faculty rather than computer science faculty.

- Students often experience friction caused by differences between standard mathematics notation and the notation used by existing functional programming languages. As one simple example, in mathematics one can write $2x$ to denote multiplication of x by 2; but many programming languages require writing a multiplication operator, for example, `2*x`. Any one such difference is small, but the accumulation of many such differences can be a real impediment to students as they attempt to move back and forth between the worlds of abstract mathematics and concrete computer programs.

For example, consider the following function defined using typical mathematical notation:

$$f : \mathbb{N} \to \mathbb{Q}$$
$$f(2n) = 0$$
$$f(2n+1) = \begin{cases} n/2 & \text{if } n > 5, \\ 3n+7 & \text{otherwise} \end{cases}$$

Now consider this translation of the function into idiomatic Haskell:

```
f :: Int -> Rational
f x
  | even x    = 0
  | n > 5     = fromIntegral n / 2
  | otherwise = 3*n + 7
  where
    n = x `div` 2
```

Although the translation may seem trivial to experienced functional programmers, from the point of view of a student these are extremely different.

DISCO is a new functional programming language, specifically designed for use in a discrete mathematics course, which attempts to solve many of these issues:

- Although DISCO is Turing-complete, it is a teaching language, not a general-purpose language. It includes only features which are of direct relevance to teaching core functional programming and discrete mathematics topics; for example, it does not include a floating-point number type. Section 2 has many examples of the language's features and some discussion of features which are explicitly excluded.
- As much as possible, the language's features and syntax mirror common mathematical practice rather than other functional languages. For example, a translation into DISCO of the example function introduced previously is shown below.

  ```
  f : N -> Q
  f(2n)   = 0
  f(2n+1) = {? n/2      if n > 5,
               3n + 7   otherwise
            ?}
  ```

  Section 2 has many more examples, and Section 3.1 discusses some notable exceptions.
- As a result—although there is as yet no data to back this up—the language should be easy for instructors to learn, even mathematicians without much prior programming experience.

DISCO is an open-source project, implemented in Haskell, with source code licensed under a BSD 3-clause license and available on GitHub.[3] Although it is possible to install DISCO locally, either from Hackage[4] or directly from source, one can also interact with DISCO in the cloud via a web browser, through the magic of replit.[5] This is the primary way that students will be instructed to use Disco, so that

[3] https://github.com/disco-lang/disco

[4] https://hackage.haskell.org/package/disco

[5] https://replit.com/@BrentYorgey/Disco#README.md

students do not need to install a Haskell toolchain or worry about exhausting the computational resources of their device. Via replit, it is entirely feasible to play with DISCO on any device with a web browser, including Chromebooks, tablets, or phones. Documentation for DISCO is hosted on `readthedocs.org`.[6]

2 DISCO by Example

We will begin by exploring some of the major features and uses of the language via a series of examples.

2.1 Greatest common divisor

Our first example is an implementation of the classic Euclidean algorithm for computing the greatest common divisor of two natural numbers, shown in Listing 1.

```
||| The greatest common divisor of two natural numbers.

!!! gcd(7,6)   == 1
!!! gcd(12,18) == 6
!!! gcd(0,0)   == 0
!!! forall a:N, b:N. gcd(a,b) divides a /\ gcd(a,b) divides b
!!! forall a:N, b:N, g:N. (g divides a /\ g divides b) ==> g divides gcd(a,b)

gcd : N * N -> N
gcd(a,0) = a                 -- base case
gcd(a,b) = gcd(b, a mod b)   -- recursive case
```

Listing 1: Definition of `gcd` in DISCO

Lines beginning with `|||` denote special documentation comments attached to the subsequent definition, similar to docstrings in Python (regular comments start with `--`). This documentation can be later accessed with the `:doc` command at the REPL prompt:

```
Disco> :doc gcd
gcd : ℕ × ℕ → ℕ

The greatest common divisor of two natural numbers.
```

Lines beginning with `!!!` denote *tests* attached to the subsequent definition, which can be either simple Boolean unit tests (such as `gcd(7,6) == 1`), or quantified properties (such as the last two tests, which together express the universal property defining `gcd`). Such properties will be tested exhaustively when feasible, or, when exhaustive testing is impossible (as in this case), tested with a finite number of randomly chosen inputs. Under the hood, this uses the `QuickCheck` [CH00] and `simple-enumeration` packages to generate inputs. For example:

```
Disco> :test forall a:N, b:N. let g = gcd(a,b) in g divides a /\ g divides b
  - Possibly true: ∀a, b. let g = gcd(a, b) in g divides a /\ g divides b
```

[6] `https://disco-lang.readthedocs.io`

```
    Checked 100 possibilities without finding a counterexample.

Disco> :test forall a:N, b:N. let g = gcd(a,b) in g divides a /\ (2g) divides b
  - Certainly false: ∀a, b. let g = gcd(a, b) in g divides a /\ 2 * g divides b
    Counterexample:
      a = 0
      b = 1
```

In the first case, DISCO reports that 100 sample inputs were checked without finding a counterexample, leading to the conclusion that the property is *possibly* true. In the second case, when we modify the test by demanding that `b` must be divisible by twice `gcd(a,b)`, DISCO is quickly able to find a counterexample, proving that the property is *certainly* false.

Every top-level definition in DISCO must have a type signature; `gcd : N * N -> N` indicates that `gcd` is a function which takes a pair of natural numbers as input and produces a natural number result. The recursive definition of `gcd` is then straightforward, featuring multiple clauses and pattern-matching on the input.

2.2 Primality testing

The example shown in Listing 2, testing natural numbers for primality via trial division, is taken from Doets and van Eijck [DE04, pp. 4–11], and has been transcribed from Haskell into DISCO. (DISCO also has a much more efficient built-in primality testing function that calls out to the highly optimized `arithmoi` package.)

```
||| ldf k n calculates the least divisor of n that is at least k and
||| at most sqrt n.  If no such divisor exists, then it returns n.
ldf : N -> N -> N
ldf k n =
  {? k            if k divides n,
     n            if k^2 > n,
     ldf (k+1) n  otherwise
  ?}

||| ld n calculates the least nontrivial divisor of n, or returns n if
||| n has no nontrivial divisors.
ld : N -> N
ld = ldf 2

||| Test whether n is prime or not.
isPrime : N -> Bool
isPrime n = (n > 1) and (ld n == n)
```

Listing 2: Primality testing in DISCO

There are a few interesting things to point out about this example. The most obvious is the use of a *case expression* in the definition of `ldf` delimited by `{? ... ?}`. It is supposed to be reminiscent of

typical mathematical notation like

$$ldf\ k\ n = \begin{cases} k & \text{if } k \mid n, \\ n & \text{if } k^2 > n, \\ ldf\ (k+1)\ n & \text{otherwise.} \end{cases}$$

However, we can't use a bare curly brace as DISCO syntax since it would conflict with the notation for literal sets (and we can't use a giant, multi-line curly brace in any case![7]). The intention is that writing `{? ... ?}` lends itself to the mnemonic of "asking questions" to see which branch of the case expression to choose. In general, each branch can have multiple chained conditions, each of which can either be a Boolean guard, as in this example, or a pattern match introduced with the `is` keyword. In fact, all multi-clause function definitions with pattern matching really desugar into a single case expression. For example, the definition of `gcd` in Listing 1 desugars to

```
gcd : N * N -> N
gcd = \p. {? a if p is (a,0), gcd(b, a mod b) if p is (a,b) ?}
```

Notice that the definition of `isPrime` uses the `and` keyword instead of `/\`. These are synonymous—in fact, `&&` and ∧ (U+2227 LOGICAL AND) are also accepted. In general, DISCO's philosophy is to allow multiple syntaxes for things with common synonyms rather than imposing one particular choice. Typically, a Unicode representation of the "real" notation is supported (and used when pretty-printing), along with an ASCII equivalent, as well as (when applicable) syntax common in other functional programming languages. Another good example is the natural number type, which can be written $\mathbb{N}$, `N`, `Nat`, or `Natural`. There are several reasons for this design choice:

- It makes code easier to *write* since students have to spend less time trying to remember the one and only correct syntax choice, or worrying about whether a particular syntax they remember comes from math class, Python, or DISCO.
- Although having many different syntax choices can make code harder to *read*, helping students learn how to interpret formal notation and how to translate between mathematics and programming notation are typical explicit learning goals of the course, so this could be considered a feature.

Notice that `ldf` is defined via currying, and is partially applied in the definition of `ld`. Just as in Haskell, every function in DISCO takes exactly one argument, but some functions can return other functions (curried style) and some functions can take a product type as input (uncurried style). Via tutorials, documentation, and the types of standard library functions, DISCO encourages the use of an *uncurried* style, since students are already used to notation like `f(x,y)` for multi-argument functions in mathematics.

Finally, this example introduces the primitive `Bool` type in addition to the natural number type `N` seen previously. DISCO also has a primitive `Char` type for Unicode codepoints, and several other numeric types to be discussed later.

2.3 Z-order

The "Morton Z-order" is one of my favorite bijections showing that $\mathbb{N} \times \mathbb{N}$ has the same cardinality as $\mathbb{N}$; it takes a pair of natural numbers, expresses them in binary, and interleaves their binary representations to form a single natural number. DISCO code to compute this bijection (and check that it really is a bijection) is shown in Listing 3.

[7] One might imagine using vertically aligned curly brace characters to simulate a giant curly brace, but that would require tricky indentation-sensitive parsing.

```
!!! forall n:N. zOrder(zOrder'(n)) == n
!!! forall p:N*N. zOrder'(zOrder(p)) == p

zOrder : N*N -> N
zOrder(0,0) = 0
zOrder(2m,n) = 2 * zOrder(n,m)
zOrder(2m+1,n) = 2 * zOrder(n,m) + 1

zOrder' : N -> N*N
zOrder'(0)    = (0,0)
zOrder'(2n)   = {? (2y,x) when zOrder'(n) is (x,y) ?}
zOrder'(2n+1) = {? (2y+1,x) when zOrder'(n) is (x,y) ?}
```

Listing 3: Morton Z-Order

This example again uses case expressions; it may seem odd to use case expressions with only one branch, but this is done in order to be able to pattern-match on the result of the recursive call to `zOrder'`. The most interesting thing about this example is its use of *arithmetic patterns*, such as `zOrder'(2n) = ...` and `zOrder'(2n+1) = ...`. This is common mathematical notation, but perhaps less common in programming languages. Any expression with exactly one variable and only basic arithmetic operators can be used as a pattern; the pattern matches if there exists a value of the appropriate type for the variable which makes the expression equal to the input. For example, the pattern `2n` will match only even natural numbers, and `n` will then be bound to half of the input.

2.4 Finite sets

DISCO has built-in *finite sets*; in particular, values of type `Set(A)` are finite sets with elements of type `A`. DISCO supports the usual set operations (union, intersection, difference, cardinality, power set), and sets can be created by writing a finite set literal, like `{1,3,5,7}`, using ellipsis notation, like `{1, 3 .. 7}`, or using a set comprehension, as in `{2x+1 | x in {0 .. 3}}`. Listing 4 shows a portion of an exercise (with answers filled in) to help students practice their understanding of set comprehensions.

Set comprehensions in DISCO work similarly to list comprehensions in Haskell (DISCO has list and bag comprehensions as well). In these examples we can see both *filtering* the generated values via Boolean guards, as well as *transforming* the outputs via an expression to the left of the vertical bar.

While on the subject of sets, it is worth mentioning that the distinction between *types* and *sets* is something of a pedagogical minefield: the distinction is nonexistent in typical presentations of mathematics, but crucial in a computational system with static type checking. This issue is discussed in more detail in Section 3.4.

One other thing this example highlights is that there is extensive, student-centered documentation available at `https://disco-lang.readthedocs.io/`. Students are pointed to this documentation not just from links in homework assignments such as this, but also by the DISCO REPL itself. Encountering an error, or asking for documentation about a function, type, or operator, are all likely to result in documentation links for further reading, as illustrated in Listing 5.

```
-- Exercise D1. For each of exA through exF below, replace the empty
-- set with a *set comprehension* so that the tests all pass, as in
-- the example.  (Remember, Disco will run the tests when you :load
-- this file.)
--
-- Some relevant documentation you may find useful:
--
-- https://disco-lang.readthedocs.io/en/latest/reference/set.html
-- https://disco-lang.readthedocs.io/en/latest/reference/comprehension.html
-- https://disco-lang.readthedocs.io/en/latest/reference/size.html
-- https://disco-lang.readthedocs.io/en/latest/reference/power.html

||| An example to illustrate the kind of thing you are supposed to do
||| in the exercises below.  We have defined the set using a *set
||| comprehension* so that it has the specified elements and the test
||| passes.

!!! example == {1, 4, 9, 16, 36}   -- test specifying 'example' elements
example : Set(N)
example = {x^2 | x in {1 .. 6}, x /= 5}  -- a set comprehension defining it

-- Now you try.

!!! exA == {1, 3, 5, 7, 9, 11, 13, 15}
exA : Set(N)
exA = {2x+1 | x in {0..7}}

!!! exD == {{1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}}
exD : Set(Set(N))
exD = {S | S in power({1..4}), |S| == 3}
```

Listing 4: Set comprehension exercise

```
Disco> :doc +
~+~ : ℕ × ℕ → ℕ
precedence level 7, left associative

The sum of two numbers, types, or graphs.

https://disco-lang.readthedocs.io/en/latest/reference/addition.html

Disco> x + 3
Error: there is nothing named x.
https://disco-lang.readthedocs.io/en/latest/reference/unbound.html
```

Listing 5: DISCO generates links to online documentation

```
import list
import oeis

-- The type of binary tree shapes: empty tree, or a pair of subtrees.
type BT = Unit + BT*BT

||| Compute the size (= number of binary nodes) of a binary tree shape.
size : BT -> N
size(left(unit)) = 0
size(right(l,r)) = 1 + size(l) + size(r)

||| Check whether all the items in a list satisfy a predicate.
all : List(a) * (a -> Bool) -> Bool
all(as, P) = reduce(~/\~, true, each(P, as))

||| Generate the list of all binary tree shapes of a given size.
!!! all([0..4], \n. all(treesOfSize(n), \t. size(t) == n))
treesOfSize : N -> List(BT)
treesOfSize(0)   = [left(unit)]
treesOfSize(n+1) =
  [ right (l,r) | k <- [0 .. n], l <- treesOfSize(k), r <- treesOfSize(n .- k) ]

||| The first few Catalan numbers, computed by brute force.
catalan1 : List(N)
catalan1 = each(\n. length(treesOfSize(n)), [0..4])

||| More Catalan numbers, extended via OEIS lookup!
catalan : List(N)
catalan = extendSequence(catalan1)
```

Listing 6: Counting trees

2.5 Trees and Catalan numbers

Listing 6 is a fun example generating and counting binary trees. It defines a recursive type `BT` of binary tree shapes, along with a function to generate a list of all possible tree shapes of a given size (via a list comprehension), and uses it to generate the first few Catalan numbers. This list is then extended via lookup in the Online Encyclopedia of Integer Sequences (OEIS) [OEI22].

The first thing to note is that DISCO has *equirecursive* algebraic types. The `type` declaration defines the type `BT` to be *the same type as* `Unit + BT*BT` (*i.e.* the tagged union of the primitive one-element `Unit` type with pairs of `BT` values). This is a big departure from the *isorecursive* types of Haskell and OCaml, where *constructors* are required to explicitly "roll" and "unroll" values of recursive types. We can see in the example that `size` takes a value of type `BT` as input, but can directly pattern-match on `left(unit)` and `right(l,r)` without having to "unfold" or "unroll" it first. Using equirecursive types makes the implementation of the type system more complex, but it is a very deliberate choice:

- There is less incidental complexity for students to stumble over. In my experience, students learn-

ing Haskell are often confused by the idea of constructors and how to use them to create and pattern-match on data types.

- DISCO has no special syntax for declaring (recursive) sums-of-products; it simply has sum types, product types, and recursive type synonyms. Of course, it would be very tedious to write "real" programs in such a language—values of large sum types like `type T = A + B + C + ...` have to be written as `left(a)`, `right(left(b))`, `right(right(left(c)))`, and so on. However, the sum types used as examples in a discrete mathematics class rarely have more than two or three summands, and working directly with primitive sum and product types helps students explicitly make connections to other things they have already seen, such as Cartesian product and disjoint union of sets. It also reinforces the *algebraic* nature of algebraic data types.

The `oeis` module is inessential, but can be a fun way for students to explore integer sequences and the OEIS. In addition to `extendSequence`, the module also provides a `lookupSequence` function, which returns the URL of the first OEIS result, if there is any:

```
Disco> lookupSequence(catalan1)
right("https://oeis.org/A000108")
```

The last things illustrated by this example are some facilities for computing with collections. The built-in `each` function is like Haskell's `map`, but works for sets and multisets in addition to lists. `reduce` is like `foldr`, but again working over sets and multisets in addition to lists. In this case, the `all` function is defined by first mapping a predicate over `each` element of a list, then reducing the resulting list of Booleans via logical conjunction. (Putting twiddles (`~`) in place of arguments is the way to turn operators into standalone functions, thus: `~/\~`.) Notice also that the `all` function is polymorphic: DISCO has support for standard parametric polymorphism.

2.6 Defining and testing bijections

Listing 7 shows part of another exercise I give to my students, asking them to define the inverse of a given function and use DISCO to check that their inverse is correct. This exercise makes essential use of the testing facility we have already seen: if a student defines a function which is not inverse to the given function, DISCO is usually able to quickly find a counterexample. Running this counterexample through the functions hopefully gives the student some insight into why their function is not correct. For example, if we try (incorrectly) defining `g2(x) = x - 1/2`, DISCO reports

```
g2:
- Certainly false: ∀x. f2(g2(x)) == x
  Counterexample:
    x = 1
```

In this example we can also see more numeric types besides the natural numbers. DISCO actually has four primitive numeric types:

- The natural numbers $\mathbb{N} = \{0,1,2,\dots\}$, which support addition and multiplication.
- The integers $\mathbb{Z} = \{\dots,-2,-1,0,1,2,\dots\}$, which besides addition and multiplication also support subtraction.
- The *fractional numbers* $\mathbb{F} = \{a/b \mid a,b \in \mathbb{N}, b \neq 0\}$, *i.e.* nonnegative rationals, which besides addition and multiplication also support division.
- The *rational numbers* $\mathbb{Q}$, which support all four arithmetic operations.

```
----------------------------------------------------------------------
-- Each of the functions below is a bijection.  Define another Disco
-- function which is its inverse, and write properties showing that
-- the functions are inverse.  Part (a) has already been done for you
-- as an example.  Part (b) has been done partially.  You should
-- complete parts (c)-(g) on your own.

-- (a) ---------------------------------------------------

f1 : Z -> Z
f1(n) = n - 5

-- EXAMPLE SOLUTION for part (a).  Definition of g1 as the inverse of
-- f1, with two test properties demonstrating they are inverse.

!!! forall z:Z. f1(g1(z)) == z
!!! forall z:Z. g1(f1(z)) == z

g1 : Z -> Z
g1(n) = n + 5

-- (b) ---------------------------------------------------

f2 : Q -> Q
f2(x) = 2x + 1

-- PARTIAL SOLUTION for part (b).  Some test properties and a type
-- declaration for g2; you should fill in a definition for g2.

!!! forall x:Q. f2(g2(x)) == x
!!! forall x:Q. g2(f2(x)) == x

g2 : Q -> Q
-- FILL IN YOUR DEFINITION HERE
```

Listing 7: Defining and testing bijections

```
Disco> :type -3
-3 : ℤ
Disco> :type |-3|
abs(-3) : ℕ
Disco> :type 2/3
2 / 3 : 𝔽
Disco> :type -2/3
-2 / 3 : ℚ
Disco> :type floor(-2/3)
floor(-2 / 3) : ℤ
Disco> :type [1,2,3]
[1, 2, 3] : List(ℕ)
Disco> :type [1,-2,3/5]
[1, -2, 3 / 5] : List(ℚ)
```

Listing 8: Numeric types and subtyping

DISCO uses *subtyping* to match standard mathematical practice. For example, it is valid to pass a natural number value to a function expecting an integer input. Mathematicians (and students!) would find it very strange and tedious if one were required to apply some sort of coercion function to turn a natural number into an integer.

These four types naturally form a diamond-shaped lattice, as shown in Fig. 1. $\mathbb{N}$ is a subtype of both

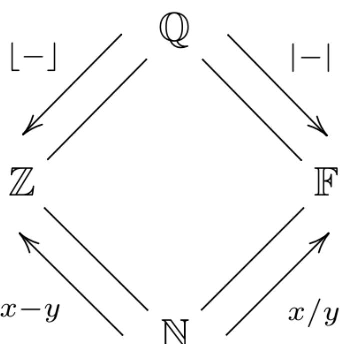


Figure 1: DISCO's numeric type lattice

$\mathbb{Z}$ and $\mathbb{F}$, which are in turn both subtypes of $\mathbb{Q}$. Moving up and left in the lattice (from $\mathbb{N}$ to $\mathbb{Z}$, or $\mathbb{F}$ to $\mathbb{Q}$) corresponds to allowing subtraction; moving up and right corresponds to allowing division. Moving down and left can be accomplished via a rounding operation such as floor or ceiling; moving down and right can be accomplished via absolute value. Listing 8 demonstrates these ideas by requesting the types of various expressions. In the last example, in particular, notice how DISCO infers the type $\mathbb{Q}$ for the elements of the list, since that is the only type that supports both negation and division.

DISCO has no floating-point type, because floating-point numbers are *the worst* [Gol91] and there is no particular need for real numbers in a discrete mathematics course.

3 Discussion and Future Work

3.1 Syntax

For the most part, DISCO tries to use syntax as close to standard mathematical syntax as possible. However, there are a few notable cases where this was deemed impossible, typically because standard mathematical syntax is particularly ambiguous or overloaded. Thinking explicitly about these cases is a worthwhile exercise, since they are likely to be confusing to students anyway.

- Mathematicians are very fond of using vertical bars for multiple unrelated things, and DISCO actually does well to allow them in many cases: absolute value, set cardinality, and the separator between expression and guards in a comprehension all can be written in DISCO with vertical bars. However, the "is a divisor of" relation is also traditionally written with a vertical bar, as in $3 \mid 21$, but DISCO does not support this notation. Including it would make the grammar extremely ambiguous. (And besides, Dijkstra would tell us that we should not use a visually symmetric symbol for a nonsymmetric relation!) Instead, DISCO provides `divides` as an infix operator. In my experience students have no problem remembering the difference.

- In mathematics, the equality symbol $=$ is also typically overloaded to denote both definition ("*let* $x = 3$*, and consider...*") and equality testing ("*if* $x = 3$*, then...*"). DISCO cannot use the same symbol for both, since otherwise it would be impossible to tell whether the user is writing a definition or entering a Boolean test to be evaluated. This is confusing for students but it seems like it can't be helped, and in any case I would argue that trying to gloss over the difference is not really doing students any favors, but simply allowing them to persist in some fundamental misunderstandings.

- DISCO allows juxtaposition to denote both function application, as in `f(3)`, and multiplication, as in `2x`. It uses a simple syntax-directed approach to tell them apart: if the expression on the left-hand side of a juxtaposition is a numeric literal, or a parenthesized expression with an operator, then it is interpreted as multiplication; otherwise it is interpreted as function application. However, this does not always get it right, and there are times when an explicit multiplication operator must be written (one notable example is when scaling the output of a function call, as in $2f(x)$; in DISCO this must currently be written `2*f(x)`, although this can hopefully be fixed). It might be worth exploring a more type-directed approach, although that would be considerably more complex. It seems like to really get this "right" requires general intelligence: for example, does the expression $f(x+2)$ denote multiplication or function application? Are you sure? How do you know? What about in the expression $x(y+2)$? Or how about "Let x be the function which doubles its argument, and consider $x(y+2)$ ..."?

3.2 Student experience

So far, I have used DISCO in my discrete mathematics course twice, in the spring semesters of 2022 and 2023. Both courses had about 25 students, mostly sophomore computer science majors, with a few mathematics majors in the mix as well. In the spring of 2023 I asked students a couple questions about Disco on the end-of-semester course evaluation. The results are shown below. Although I did not get a good response rate and the results have no statistical significance, they are at least encouraging. The few students who wrote optional textual comments also had very positive things to say.

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Using Disco helped me learn the mathematical ideas in this course better.	0	2	0	7	7
Learning Disco helped me improve my computer programming skills.	0	0	4	7	5

3.3 Type system

DISCO and its type system were designed to be intuitive for students and to corresponding closely to mathematical practice, but this has not always led to the simplest type system from an implementation point of view!

- As previously mentioned, DISCO has *subtyping* in order to accommodate typical mathematical practice. DISCO's subtyping is *structural*, meaning that we only really need concern ourselves with subtyping relationships between primitive types; a subtyping relation between complex types (for example, sum, product, or function types) can always ultimately be broken down into subtyping relations between simpler types. Subtyping complicates the type system since, for example, when typechecking the application of a function to an argument, we cannot just check that the types match via unification, but we must instead emit a subtyping constraint which we check later.
- DISCO also has parametric polymorphism, since a language without polymorphism would not really give students a good idea of the expressive power of statically typed functional programming. Of course, this means that typing constraints can involve unification variables as well as *skolem variables* (when *checking* a polymorphic type).
- DISCO's type system must actually support *qualified* polymorphism (similar to Haskell's type classes, but with only a specific set of built-in classes) in order to be able to infer types in a setting where some types support certain operations (*e.g.* subtraction or division) and some do not. For example, what is the type of `\x. x - 2`? Most generally, this function has a type like $\forall a.\ (sub(a), \mathbb{Z} <: a) \Rightarrow a \to a$, that is, is a polymorphic function with type $a \to a$ for any type a which supports subtraction and has $\mathbb{Z}$ as a subtype, *i.e.* either $\mathbb{Z}$ or $\mathbb{Q}$. (As a nice exercise, you might like to convince yourself that none of $\mathbb{Z} \to \mathbb{Z}$, $\mathbb{Z} \to \mathbb{Q}$, $\mathbb{Q} \to \mathbb{Z}$, or $\mathbb{Q} \to \mathbb{Q}$ will work—some of them are invalid types for the function, and some of them, although valid, are not general enough.)

  Such types are currently only allowed internally, during type inference, but must be monomorphized away before showing types to users. This is sound, but can be rather confusing. For example, DISCO will report that the type of `\x. x - 2` is $\mathbb{Z} \to \mathbb{Z}$, but will also happily allow it to be applied to a fractional input such as $5/2$, which would be a type error if its most general type were really $\mathbb{Z} \to \mathbb{Z}$.

  ```
  Disco> :type \x. x - 2
  λx. x - 2 : ℤ → ℤ
  Disco> (\x. x - 2) (5/2)
  1/2
  ```

One interesting idea to improve the situation would be to show the user *multiple* potential monomorphic instantiations of a general type scheme, something like this, perhaps:

```
Disco> :type \x. x - 2
λx. x - 2
  : ℤ → ℤ
  : ℚ → ℚ
```

- As mentioned before, DISCO has equirecursive types. The big wrinkle this adds to the type system is that simple structural equality (or unification) no longer suffices; when recursive type synonyms are involved, two types can be the same even though they look different.

The combination of qualified parametric polymorphism, subtyping, and equirecursive types makes for an overall system which seems only barely on the edge of tractability. For the implementation of type inference and checking I relied heavily on Traytel *et al.* [TBN11] who describe the implementation of a similar type system for Isabelle/HOL. There are almost certainly bugs, but overall I am fairly confident in the soundness of the type system.

3.4 Types vs sets

Axiomatic set theory is usually taken as the *de facto* foundation for mathematics. On the other hand, in practice, mathematicians usually behave more as if they were working in some kind of type-theoretic foundation, which makes a statically typed functional language a good match for mathematics as it is practiced (for example, see HoTT [Uni13] and Lean [MU21]). However, one area where there seems to be a big mismatch is in the distinction between *types* and *sets*.

To most mathematicians and every discrete mathematics textbook ever, $\{2,4,7\}$ and $\mathbb{N}$ are both examples of *sets*. The former is finite and the latter (countably) infinite, but they are both fundamentally the same kind of thing, and it makes sense to talk about (for example) their difference, $\mathbb{N} - \{2,4,7\}$, which is also a set. In DISCO, however, $\{2,4,7\}$ and $\mathbb{N}$ are very different things: the former is a value of type `Set(ℕ)`, whereas the latter is a type, and $\mathbb{N} - \{2,4,7\}$ is so nonsensical that it is a *syntax* error! One might reasonably wonder: why the mismatch? Why not try to make DISCO more closely align with common mathematical practice, in accordance with DISCO's stated philosophy?

Although conflating sets and types might be fine on a theoretical level (at least, as long as one does not worry about deeper foundational issues), on a practical level it introduces several big problems:

- The ability to use arbitrary finite sets as types would lead to what is essentially a system of *refinement types*. Although this is well-studied and has many practical motivations, it quickly leads to undecidable typechecking, the need for tools like SMT solvers, and the requirement for users to provide annotations to help the system understand why a given type is valid. For example, to typecheck `f : ℕ -> {2,3,7}` would require somehow checking that for any natural number input, the function `f` will always return either 2, 3, or 7, which could depend on complex reasoning about the behavior of the function. Calling out to an SMT solver in order to typecheck a teaching language to be used by students seems like a non-starter.
- Conversely, the ability to use types as value-level sets introduces all sorts of difficulties, chief among which is the fact that most types correspond to *infinite* sets. Set values would have to be represented at runtime as some kind of abstract set expressions rather than simply as sets of values, and operations like membership checking become only *semi*-decidable at best. What's more, these infinite set values would not really correspond to their supposed mathematical counterparts in some

> subtle ways. For example—as we often teach discrete mathematics students—the power set of the natural numbers is uncountable; but if `N` were usable as a value of type `Set(N)` in DISCO, then `power(N)` would actually represent the set of *computable* subsets of $\mathbb{N}$, which is countable!

The one slight blurring of categories which seems both feasible and desirable would be the ability to use finite set values as domains for $\forall$ and $\exists$ property quantifiers, so one could write, for example, `forall x in [0..10]. x^2 <= 100`. It seems theoretically straightforward to incorporate such finite sets into the existing machinery for property checking, though this has not been done yet.

In any case, how should we present and explain the relevant distinctions to students? Honestly, I'm not entirely sure. My best approach at the moment revolves around two ideas:

- First, explain to students that DISCO can only represent *finite* sets. This is easy enough to understand: if we allowed infinite sets, then certain operations might require infinitely long computations.
- We can then explain that types can be thought of as a particular collection of "ur-sets" out of which we can build and carve out all other sets. For particularly keen students, we can explain that types are sets with particularly nice structural properties. For example, $\mathbb{N}$ is the unique set that includes 0 and is closed under the successor operation; in contrast, there is no nice structural way to define $\{2,4,7\}$ besides just listing its elements. These nice structural properties are precisely what enable decidable typechecking without having to resort to SMT solvers.

3.5 Formal proofs

A discrete mathematics course often includes an introduction to proof writing. Currently, Disco only helps with this indirectly: it helps students practice expressing themselves formally, and property-based testing can give early feedback to see whether a particular proposition is worth trying to prove. Ideally, however, there would be a way to use Disco more directly in constructing formal proofs, or a way to integrate it with existing tools for doing so.

3.6 Error messages

When the DISCO project first started, I had grand designs for the way the system would interact with the user in the case of type errors [YEEI18]. Unfortunately, partly because I was intimidated by my own grand designs, and partly because error messages are hard, the system currently does not have very good error messages! For example, here is a terribly uninformative one:

```
Disco> each(3, [1,2,3])
Error: the shape of two types does not match.
https://disco-lang.readthedocs.io/en/latest/reference/shape-mismatch.html
```

In practice, I just tell students to ask me for help when they run into errors they can't figure out, but this obviously limits wider adoption. Improving error messages will be another big focus for work in the upcoming year.

4 Acknowledgments

DISCO was originally born out of a conversation with Harley Eades at TFPIE in 2016, and I'm very grateful to Harley for those initial conversations and many good ideas. I am also grateful to the many

people who have contributed to the DISCO codebase over the years, including my students Callahan Hirrel, Bosco Ndemeye, Sanjit Kalapatapu, Jacob Hines, Eric Pinter, and Daniel Burnett, as well as other collaborators including Shay Lewis, Ryan Yates, Tristan de Cacqueray, and Chris Smith.

References

[CH00] Koen Claessen & John Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pp. 268–279, doi:10.1145/351240.351266.

[CM13] Association for Computing Machinery (2013): *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*, doi:10.1145/2534860. Available at http://www.acm.org/education/CS2013-final-report.pdf.

[DE04] Kees Doets & Jan van Eijck (2004): *The Haskell Road to Logic, Maths, and Programming*. Texts in Computing. King's College Publications.

[Gol91] David Goldberg (1991): *What every computer scientist should know about floating-point arithmetic*. *ACM computing surveys (CSUR)* 23(1), pp. 5–48, doi:10.1145/103162.103163.

[Hen02] P. B. Henderson (2002): *Functional and declarative languages for learning discrete mathematics*. Technical Report 0210, University of Kiel.

[Lib07] Liberal Arts Computer Science Consortium (2007): *A 2007 Model Curriculum for a Liberal Arts Degree in Computer Science*. *J. Educ. Resour. Comput.* 7(2), doi:10.1145/1240200.1240202.

[MU21] Leonardo de Moura & Sebastian Ullrich (2021): *The Lean 4 theorem prover and programming language*. In: *International Conference on Automated Deduction*, Springer, pp. 625–635, doi:10.1007/978-3-030-79876-5_37.

[OEI22] OEIS Foundation Inc. (2022): *The On-Line Encyclopedia of Integer Sequences*. Available at http://oeis.org/.

[OHP06] John O'Donnell, Cardelia Hall & Rex Page (2006): *Discrete Mathematics Using a Computer*. Springer-Verlag London.

[SW02] C. Scharff & A. Wildenberg (2002): *Teaching discrete structures with SML*. Technical Report, University of Kiel.

[TBN11] Dmitriy Traytel, Stefan Berghofer & Tobias Nipkow (2011): *Extending Hindley-Milner type inference with coercive structural subtyping*. In: *Asian Symposium on Programming Languages and Systems*, Springer, pp. 89–104, doi:10.1007/978-3-642-25318-8_10.

[Uni13] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[Van11] Thomas VanDrunen (2011): *The Case for Teaching Functional Programming in Discrete Math*. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, ACM, New York, NY, USA, pp. 81–86, doi:10.1145/2048147.2048180.

[Van13] Thomas VanDrunen (2013): *Discrete mathematics and functional programming*. Franklin, Beedle & Associates Incorporated.

[Van17] Thomas VanDrunen (2017): *Functional Programming as a Discrete Mathematics Topic*. *ACM Inroads* 8(2), p. 51–58, doi:10.1145/3078325.

[Wai92] Roger L. Wainwright (1992): *Introducing Functional Programming in Discrete Mathematics*. In: *Proceedings of the Twenty-third SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '92, ACM, New York, NY, USA, pp. 147–152, doi:10.1145/135250.134540.

[Xin08] Cong-Cong Xing (2008): *Enhancing the Learning and Teaching of Functions Through Programming in ML*. *J. Comput. Sci. Coll.* 23(4), pp. 97–104, doi:10.5555/1352079.1352096.

[YEEI18] Brent A. Yorgey, Richard A. Eisenberg & Harley D. Eades III (2018): *Explaining Type Errors*. In: *Off The Beaten Track*. Available at https://popl18.sigplan.org/details/OBT-2018/8/Explaining-Type-Errors.