Teaching How to Program using Automated Assessment and Functional Glossy Games (Experience Report)

JOSÉ BACELAR ALMEIDA, Universidade do Minho and INESC TEC, Portugal
ALCINO CUNHA, Universidade do Minho and INESC TEC, Portugal
NUNO MACEDO, Universidade do Minho and INESC TEC, Portugal
HUGO PACHECO, Universidade do Minho and INESC TEC, Portugal
JOSÉ PROENÇA, Universidade do Minho and INESC TEC, Portugal

Our department has long been an advocate of the functional-first school of programming and has been teaching Haskell as a first language in introductory programming course units for 20 years. Although the functional style is largely beneficial, it needs to be taught in an enthusiastic and captivating way to fight the unusually high computer science drop-out rates and appeal to a heterogeneous population of students.

This paper reports our experience of restructuring, over the last 5 years, an introductory laboratory course unit that trains hands-on functional programming concepts and good software development practices. We have been using *game programming* to keep students motivated, and following a methodology that hinges on *test-driven development* and *continuous bidirectional feedback*. We summarise successes and missteps, and how we have learned from our experience to arrive at a model for comprehensive and interactive functional *game programming assignments* and a general functionally-powered *automated assessment platform*, that together provide a more engaging learning experience for students. In our experience, we have been able to teach increasingly more advanced functional programming concepts while improving student engagement.

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

Additional Key Words and Phrases: programming education, automated assessment, test-driven development, gamification

ACM Reference Format:
José Bacelar Almeida, Alcino Cunha, Nuno Macedo, Hugo Pacheco, and José Proença. 2018. Teaching How to Program using Automated Assessment and Functional Glossy Games (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 82 (September 2018), 17 pages. https://doi.org/10.1145/3236777

1 INTRODUCTION

Following recent trends [Impagliazzo et al. 2016], the number of students enrolling in the *Computer Science Engineering* (CSE) course at the University of Minho, Portugal, has been steadily rising over the last decade. It is since 2016 the largest course at the university, currently receiving up to 160 students each year. This increase could be justified by the growing need of computer scientists in our society [Impagliazzo et al. 2016], and by the wider range of students regarding gender, skills, and personalities enrolling in the course. However, our own experience, corroborated by recent studies [Pappas et al. 2016], shows that computer science (CS) education still presents

Authors' addresses: José Bacelar Almeida, Universidade do Minho , INESC TEC, Portugal; Alcino Cunha, Universidade do Minho , INESC TEC, Portugal; Nuno Macedo, Universidade do Minho , INESC TEC, Portugal; Hugo Pacheco, Universidade do Minho , INESC TEC, Portugal; José Proença, Universidade do Minho , INESC TEC, Portugal.

2475-1421/2018/9-ART82
https://doi.org/10.1145/3236777

unusually high drop-out rates, with the first year being critical. It is thus essential for introductory programming course units to engage and motivate students, which directly reflects their academic success [Bergin and Reilly 2005].

The large body of research on teaching introductory programming [Pears et al. 2007] agrees that units should be language-agnostic, focus on elementary programming concepts, and develop problem solving skills. Paradigmatically, mainstream object-oriented and imperative programming languages are still the most widely used in education, even though they are often criticised for being especially unsuited for novices [Pears et al. 2007]. As members of the functional programming (FP) community, our group has long been sympathetic of a functional-first approach [Chang et al. 2001] and has – not alone [HaskellWiki 2018] – been teaching Haskell in introductory programming units for 20 years. The clean syntax and semantics of modern functional languages are a good fit for students' knowledge of high-school algebra and allow the introduction of elementary programming techniques without unnecessary details regarding lower-level abstractions. The tight interplay between theory and practice provides a good motivation to reason formally about programs early in the curriculum [Chakravarty and Keller 2004]. Moreover, the high language expressiveness allows to tackle complicated challenges early on, with the type system playing a crucial role of aiding students to structure the problem solving process [Tirronen et al. 2015].

To keep the CS course aligned with the current panorama, we have been restructuring, since 2013, an introductory programming unit that uses Haskell as a first programming language and covers a wide range of software development practices. This paper summarises our experience with this 5-year process, that faced the following challenges: (i) targeting *fresh students* in their first programming unit, so no prior knowledge in CS can be assumed; (ii) *many students* enrolled, over 200, and a teaching team of up to 7 staff members and several TAs; (iii) *high drop-out rates* in introductory CS, which reassert the need to appeal to a young, heterogeneous student population.

To address these challenges we focused on providing the students assignments and automated feedback that yielded gratifying visual results and stimulated healthy competition, while targeting relevant software development skills. The outcomes of this process are twofold: first, an assignment model based on game development and the *gamification* of tasks, that is instantiated with a concrete subject each year, and second, a publicly available *Automatic Assessment Platform* (AAP) for Haskell projects – HAAP– that supports the students throughout the development and helps instructors keep track of their progress. The rest of this section details our perspective on these two topics, and the remainder of the paper reports on the evolution of these artefacts so that our experience can be replicated by other members of the FP community.

1.1 Using Games as Assignments

Many introductory CS units have been using games in the classroom to improve student engagement, motivation and learning. Instructors have been conveying programming concepts: through game programming [Bayliss and Strout 2006; Haden 2006; Sung et al. 2008]; through game playing and visualization [Barnes et al. 2008; Li and Watson 2011]; or by including game elements such as scoring, story or competition [Iosup and Epema 2014; Powers et al. 2007]. This is more appealing to the new "multimedia" generations of students, familiar with the world-wide-web, elaborate graphical user interfaces (GUIs), and used to fast-paced interactivity.

This is not particularly new, and we can also identify a trend to develop graphical video games or animations in introductory FP units [Achten 2008; Felleisen et al. 2009; Lüth 2003; Morazán 2010]. Furthermore, we argue that functional video game programming constitutes a particularly sweet spot. Mentors can develop powerful abstractions that allow students to easily start building game-like applications by focusing on problem solving in a familiar domain, liberating students from having to deal with side effects or to understand all the underlying graphics and user interactions. In

this context, the type system becomes instrumental to guide and restrict student behaviour [Crestani and Sperber 2010]. Moreover, the assignments can explore competitive elements, inviting students to incrementally develop game strategies that particularly benefit from FP.

Unlike most previous experiments, which target more experienced students or focus on only certain parts of a game, our students are asked to build a complete game from scratch in their first encounter with a programming language. This is made possible through the use of Gloss [Lippmeier 2010], an excellent real-time animation library for Haskell that liberates beginner programmers from the intricate IO-driven aspects of animation, allowing them to focus on the pure functional logic of the game to quickly produce impressive visuals and interactive solutions.

1.2 Automated Assessment Platforms (AAP)

To maintain motivation and comfort levels, students also need proper guidance and frequent feedback as early as possible, so that they can learn to understand and fix defects in their programs by forming hypotheses and experimentally testing them [Bergin and Reilly 2005]. To provide timely feedback to this many students (over 200), it is imperative to rely on machine support, and previous experiences [Sergey 2016] suggest that FP is a good fit for the automatic grading and ranking of large-scale student competitions. HAAP, the in-house platform developed for this unit, aims at seamlessly promoting good software development practices by: being flexible in the *assessment* of program correctness; providing automatic, frequent and informative *feedback*, both graphical and detailed; supporting the definition of personal *test-scenarios*; *reporting* on static and dynamic quality metrics; and being integrated with *version control* systems.

Up to our knowledge, there is no off-the-shelf framework with the required flexibility and heterogeneity to seamlessly adapt to our context. Initially, we relied on Mooshak [Leal and Silva 2003], a web-based system for managing programming contests, but over time it revealed to be inflexible for FP and incompatible with good software development practices: instructors could only score submissions by defining unit tests as pairs of input/output text files or by writing command-line assessment scripts; students only had access to their score; and submitting single code files through a web GUI hindered the modularity of solutions and caused unnecessary overheads. This discouraged students to write their own tests and failed to provide informative feedback, which resulted in competitive students getting stuck on small technical details that were hard to debug. While other successful AAPs have been proposed [Ihantola et al. 2010], the effort required to integrate our tight FP-powered project would amount to developing the same FP-flavoured features offered by HAAP and orchestrating them with a more general development environment.

Given our long experience with FP, we were able to capitalize on the power of several existing open-source libraries and tools to build HAAP, greatly improving the comprehensiveness and quality of our assessment and feedback while adhering to best practices. HAAP is a plugin-driven platform that supports the construction of: 1) a *web frontend* with comprehensive feedback to students, including functional correctness, internal quality metrics, test validity, rankings, visual representations and replays; and 2) a *backend* that periodically assesses student submissions and generates reports for both student feedback and grading. Its architecture allows instructors to instantiate HAAP to each year's assignment by simply composing and orchestrating generic plugins.

From this didactic experience we perceived that, on one hand, the development of interactive games with open and competitive tasks can increase students' motivation, and consequently the time and energy invested by them, and on the other hand, timely and rich feedback can aid both students on assimilating the concepts and instructors on teaching and assessment.

Organisation of the paper. Section 2 presents the overall structure of our programming unit, followed by a detailed explanation on the evolution of the model of the assignment given to students

Table 1. Curriculum of the FPro and PLab units. *Since 17/18, previously taught in week 8.

Week	1	2	3	4	5	6	7	8	9	10	11	12	13
FPro	Basic types: primitives, tuples	Basic Functions: equations, conditions	*Non-recursive inductive data types**	Lists & recursive functions	Lists: sorting algorithms	Higher-order functions: `map`, `(.)`	Higher-order functions: folds	Recursive types: lists, trees	Binary search trees: search, insertion	Type classes: `Eq`, `Ord`	Type classes: `Show` IO: motivation	IO: usage examples	Revisions
PLab	Development environment	Developing Haskell programs: GHC(i)	Code documentation: Haddock	Project introduction: Phase 1 + Testing	Version control systems: SVN	Project solving: Task 1	Project solving: Task 2	Project solving: Task 3	Project introduction: Phase 2 + Quality	Gloss: Demo program	Project solving: Phase 2	Technical reports: LaTeX	Project solving: Phase 2

in Section 3. Section 4 presents the developed HAAP and one concrete instantiation, Section 5 discusses lessons learned and Section 6 concludes this paper.

2 COURSE CURRICULUM

Our university has a two-decades-long tradition of teaching FP as an introductory first semester unit. In 07/08, the transitory year of the CSE course into the Bologna Process,[1] this introductory unit has been split into two, *Functional Programming* (FPro) and *Programming Laboratories* (PLab), each worth 5 ECTS. The FPro unit is comprised of a 2-hours lecture class and a 2-hours practical class, weekly. Its goal is to teach basic FP concepts, using the Haskell language, as well as promote basic algorithmic reasoning (Table 1). Evaluation is done via two written exams. In contrast, the PLab unit is comprised of a 2-hours laboratory class, where the students practice hands-on the concepts acquired in FPro and basic technical skills, learning how to use software development tools and the Haskell ecosystem (Table 1). Students are assessed through the pairwise development of a medium-sized Haskell project, and by their performance and engagement in the classes. Besides the regular classes, students can attend 4 hours of joint study sessions with the faculty staff and TAs involved in these two units (which overlaps, including the authors of this report).

Our teaching approach follows the method of Vihavainen et al. [2011], that focuses more on the learning process rather than on personal skill and has been applied to introductory programming course units with positive results [Vihavainen et al. 2013], by emphasising the following values: the student will only master the subject by *doing it*; there is *continuous bidirectional feedback*, with the student receiving feedback on his progress while the instructor monitors his problems and challenges; there is *no compromise* and the student can take as long as necessary. The method is applied in coordination among both the FPro and PLab units.

FPro. The method starts with a lecture in which the instructor builds a conceptual model of each programming technique. In the practical class, the instructor assumes a more active role and solves a few exercises together with the class, while explaining his decisions and self-correction policies during the process. These exercises are sufficiently abstract to foster abstract thinking and problem solving, and to avoid encumbering students with precise Haskell syntax and compile errors [Tirronen et al. 2015]. Students are then asked to apply the techniques by solving small,

[1] The Bologna Process is a joint European agreement (http://www.ehea.info) to ensure compatibility and coherence of higher-education qualifications, centred on the ECTS (European Credit Transfer and Accumulation System) unit classification.

incremental exercises covering the topics taught in the lecture. Up to 17/18, inductive data types were only introduced in week 8. (This forced the first tasks of the PLab project to rely solely on textual representations and primitive data types, which we felt biased the students' mindset against developing richer data structures later on.) This topic has thus been pulled back to week 3.

After 4 weeks there is a simple eliminatory written test to assess whether students have acquired basic FP skills. Its purpose is to motivate continuous study from the beginning, and the exercises that may appear are published in the first week. The final exam evaluates every topic of the unit.

PLab. Complementing FPro, the PLab unit is comprised solely of hands-on laboratory classes, where students collaborate in groups of two [Nagappan et al. 2003] and are encouraged to solve the tasks posed by the project. The tasks are described in a way that allows the students to incorporate the knowledge acquired in FPro throughout the semester. The instructors teach mainly present the technical tools needed to develop the project, and actively aid the students in understanding the assignment and building the pieces of the puzzle. The students are also taught how to write tests, run the tools manually in a command line environment and carefully interpret their feedback. The various tasks become incrementally more open, to gradually fade the instructor-provided support, and students are expected to arrive at different solutions depending on their level of investment and knowledge.

An important part of our mentoring process is to provide timely and accurate feedback to students via an AAP [Vihavainen et al. 2013] that periodically generates comprehensive visual and textual feedback on the correctness of the tasks and the validity of the test-scenarios. Although the feedback is more frequent and detailed than what an instructor could reasonably perform, too much (or inadequate) feedback can lead to trial-and-error programming. To encourage students to identify problems and think about the correctness of their code from the beginning, the project promotes *test-driven development* (TDD) [Edwards 2003a] by having students write their own test-scenarios. The AAP only evaluates the submitted solutions against the oracle for the students' own test-scenarios, what has been shown to lead to higher quality code [Edwards 2003b]. During laboratory classes, instructors monitor the feedback reports to better understand students' difficulties. The AAP also gives feedback regarding other software development skills that the students are expected to acquire in the unit, including the style/quality of their code, the thoroughness of their test-scenarios and documentation, and their effective collaborative use of version control systems.

3 PROGRAMMING LABORATORIES PROJECT

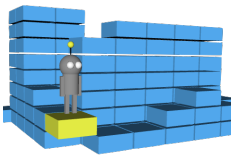
In the 13/14 academic year, we started reformulating the PLab unit and the model for its project assignment around the theme of designing and implementing a small interactive grid-based game. This is a nice short exercise for an experienced programmer, but challenging to do from scratch for beginners who are learning the basics of (functional) programming in general. In the first classes, the instructor teases the students with a complete game and some student submissions from the previous years, and presents the project by highlighting how the different tasks arise as necessary components for putting a game together. This reassures students of the relevance of each task, and gives them high hopes of developing a full functional game as a first programming experience.

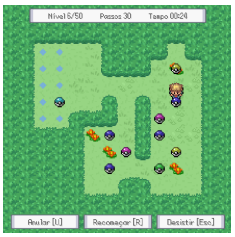
3.1 History

Our project conceptualisation and the tasks expected from students have evolved over the years:

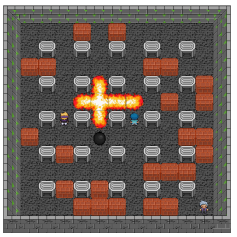
13/14 ***Carcassonne board game*** Implement discrete game logic and design playing strategies. XML-based board representation.

14/15 ***Lightbot educational game*** Implement discrete game logic, design robot strategies and animate an execution of the puzzle. Textual board representation.


X3dom *Lightbot* 14/15.



Gloss *Sokoban* 15/16.


Gloss *Bomberman* 16/17.

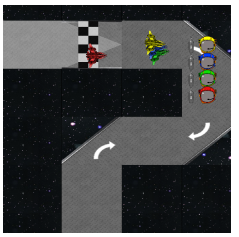

Gloss *MicroMachines* 17/18.

Fig. 1. Game artefacts designed by PLab students.

15/16 *Sokoban puzzle game* Implement discrete game logic, and interactive GUI. Textual board representation.

16/17 *Bomberman battle game* Implement discrete game logic, design bot strategies and interactive GUI. Textual board representation.

17/18 *MicroMachines racing game* Implement continuous game logic, design bot strategies and interactive GUI. Inductive data type track representation.

These assignments have emphasised and rewarded good software development practices: testing, documentation, coding style, SVN usage, and LaTeX reports. From 13/14 to 17/18, we have progressed from asking students to manipulate text-based game states and no GUI design, to having them design complete animated games in Gloss, as exemplified in Fig. 1 by screenshots of student submissions.

Hall of Fame. A more detailed account on the evolution of the assignment can be found in the interactive PLab hall of fame available online,[2] that includes a playable compilation of the best student submissions and replays of student competitions.

3.2 Functional Game Development in Gloss

Functional Reactive Programming (FRP) [Courtney et al. 2003] provides an elegant functional approach to graphical design, via a modular, declarative description of animated objects using continuous signals (e.g., time) and discrete events (e.g., keystrokes). However, its abstract style tends to be too complicated for beginners. Gloss is an FRP-inspired Haskell library developed by Ben Lippmeier [2010] that uses OpenGL under the hood and provides just the right level of functionality for beginners to develop 2D games. Gloss and its relatives [Breitner and Smith 2017; Felleisen et al. 2009] have been gaining popularity among the Haskell community to interactively teach mathematics and introductory CS from middle-schoolers to high-school students.[3]

Designing a simple animated game in Gloss starts by defining a data type to model the *game state* and setting its initial value:

```
type GameState = ...
start :: GameState
```

The state shall include all game (boards, players, items) and screen (leaderboards, menus) information. The animation of the game is defined by *three pure functions*. The *first* draws the state on the screen by translating a **GameState** into an inductive data type that represents 2D **Pictures**:

```
draw :: GameState → Picture
```

[2] haslab.github.io/Teaching/LI1/

[3] Unlike [Breitner and Smith 2017; Felleisen et al. 2009], we do not consider distributed games.

A **Picture** can contain geometrical figures such as lines, polygons or circles, or even external images. The *second* function describes how the game reacts to events, e.g., the press/release of a key, mouse motion or window resize:

```
reactEvent :: Event → GameState → GameState
```

This function shall be used to implement the logic of the purely discrete elements of the game, e.g., the impact of pressing a key to move in a certain direction. The *third* function describes how the game evolves over time, receiving the time elapsed (in fractions of seconds) since the last call:

```
reactTime :: Float → GameState → GameState
```

This can be used to program continuous events, e.g., timed bomb explosions or car movement. Note that, from the programmer's perspective, the game reacts independently to **Event**s and time, what helps separating concerns. The Gloss `play` function puts everything together:[4]

```
play :: ... → w → (w → Picture) → (Event → w → w) → (Float → w → w) → IO ()
```

Note that `play`, as a general game interface, is polymorphic on worlds. Here, we use our **GameState**:

```
game = play ... start draw reactEvent reactTime
```

And our game is complete! Gloss neatly hides complicated graphics, event handling, parallelism and side effects from the programmer. To maintain the game logic purely functional, any required IO data, such as reading image files, should be done *a priori* and stored in the **GameState**.[5]

3.3 Model of the Student's Project

We now present a model of the project, cumulatively refined throughout our 5-year experiment, that we believe best embodies the method exposed in Section 2. The model is general enough to target any grid-based arcade game, as will be illustrated through game elements used in different years. The model is organised into 2 phases, each containing 3 tasks and an artefact delivery submitted through the group's SVN repository. After the final delivery, students have to submit a written LaTeX report and personally present their work to the instructors. The model is designed for integration with Gloss and tasks are designed to be independent and engaging on their own, catering to different student profiles. All tasks amount to defining pure functions satisfying a provided type contract. The last task of each phase is a competitive open task designed to foster students' creativity and problem solving skills: they are given a trivial solution that ranks last, and can iteratively invest as much as they want and attempt various techniques to improve their ranking. Due to the open nature of these tasks, it is not clear beforehand what will be the average ranking achieved by the students. The instructors also contribute with solutions of varying complexity, clearly identified and to be used as reference by students. To simulate game levels and keep students engaged, instructors can gradually submit more "competitive" solutions to the leaderboards as the top students achieve a certain ranking. To promote TDD, students are asked to design test-scenarios (i.e., examples of arguments to run their functions) for all non-competitive tasks.

Task 1: State representation. The goal of the first task is to familiarise the students with simple data types and programming constructs, and involves manipulating the internal state of the game that is used throughout the succeeding tasks. This state is encoded as a grid of positions plus some additional information (e.g., player and item positions). To establish a common interface, the **State** is modelled as a simple non-recursive inductive data type and provided by the instructors.[6] The students are expected to process this **State** using only basic functions, while taking into

[4] The dotted arguments are general parameters such as window size and animation frame rate that can be fixed for students.
[5] Advanced programmers can also use monadic variants of Gloss library functions that allow interleaving IO actions.
[6] Until 16/17 the state in Phase 1 involved only primitive types since inductive data types were presented later in FPro.

consideration the game's context (e.g., board shape or kinds of items). Task 1 addressed conversion from another structured format (e.g., XML in 13/14), validation (14/15 and 15/16), or generation (16/17 and 17/18). For students, this amounts to writing functions with signatures similar to:

```
convert  :: XML → State          -- 13/14
validate :: State → Bool         -- 14/15, 15/16
generate :: Int → Int → State    -- 16/17
build    :: [Command] → State    -- 17/18
```

Simultaneously, the students are introduced to the software development process, for instance the proper use of version control systems. In particular, all code that is to be evaluated must be kept up-to-date in the group's SVN repositories.

Task 2: Core game logic. This task asks the students to start implementing the functional layer of the game. This involves encoding some of the game rules as a step-wise function that reacts to external commands (13/14 – 16/17) or navigates a grid (17/18):

```
next :: State → Command → State   -- 13/14 - 16/17
step :: State → Marker → Marker   -- 17/18
```

The students are given the necessary data types, such as `Command`s or `Marker`s, which capture the information needed to compute the updated state or location. Due to the richness of the input data types and the relative complexity of the functions, students start to notice at this point the importance of defining test-scenarios with proper coverage, to consider less obvious situations (e.g., players moving into a variety of different tile types).

Task 3: Open competition. The goal of this open task is to train students' problem solving skills, and does not preclude any particular approach. It should be designed so that solutions can be ranked, and that better-ranked solutions are incrementally harder to achieve. In 16/17 the task was to implement compression/decompression functions for efficiently saving the game state mid-game:

```
encode :: State → String          decode :: String → State -- 16/17
```

Any solution where `decode ∘ encode = id` would be correct, but a good rank required identifying redundant information. In 17/18 the task was to approximate the location of a player after a given amount of time, ignoring external forces (such as gravity, friction, and acceleration).

```
move :: State → Player → State -- 17/18
```

The main challenges were the use of real numbers to denote locations and time, and the fact that players could bounce off walls. A trivial implementation consisted of multiplying time by velocity, but a good rank required correctly implementing laws of physics and considering multiple bounces.

Code from inexperienced students can be difficult to follow and fairly assess. This is alleviated by constantly reminding and encouraging them of the importance of thoroughly documenting code, using special annotations to be analysed by the popular Haddock tool to generate documentation.

Task 4: Full game logic. The goal of this task is to complete the logic of the game, by integrating previous tasks and enhancing them with missing functionality. Typical functionalities requested at this stage involve handling timed events, such as deciding how exploding bombs evolve with time, or how a car moves over time. This amounts to implementing a function that updates the state after one 'tick' of the game (16/17), or after an elapsed period of time (17/18):

```
tick   :: State → State        -- 16/17
elapse :: State → Time → State -- 17/18
```

This task also helps students realize the importance of writing modular code, and reusing functionalities developed in the previous tasks without code duplication. At this point, students are encouraged to use higher-order functions simultaneously introduced in FPro.

Task 5: Animate the game. In this task, the students finally put together the pieces to build a complete playable game (15/16 – 17/18). This requires minimal adaptation to connect Tasks 2, 3 and 4 to the Gloss `reactEvent` and `reactTime` functions, and the main task is indeed to implement the `draw` function that visualizes a **`GameState`**.[7] To exercise the definition of rich data types, which up to this task had been defined by the instructors, students are asked to model a new **`GameState`** that includes the previous **`State`** information, extended with extra information required to draw the GUI (e.g., game menus, control levels, include a timer, or temporary visual effects) or with additional functionalities. In fact, the quality of their GUI is greatly bounded by the richness of their enhanced **`GameState`**. The need to manipulate Gloss **`Picture`**s also guarantees that students will train functions over recursive inductive data types (besides lists).

Initializing the `game` allows students to train basic IO functionalities, e.g., for loading game sprites before launching the game. Eager students can explore more advanced game features (e.g., playing sounds or using randomness) that require using monadic versions of the base Gloss functions.

Task 6: Final tournament. Having a fully playable game, the students are then asked to encode automatic strategies for winning it (13/14, 14/15, 16/17 and 17/18). Students are encouraged to exercise a more abstract thinking in order to transpose game strategies into programming algorithms. For them, this amounts to writing a function that, given a state and some additional information like whose turn it is, computes a **`Command`** with the next play:

```
play :: State → ... → Command
```

In contrast to Task 3, in this task the students directly compete against each other and are able to see animated replays. Students are also encouraged to play against their own strategies, for gratification and debugging, by integrating them into their implementation of the game (Task 5). At this point, they are likely to encounter complexity and performance issues, as it is easy for ill-thought-out strategies to time out or run out of resources.

4 HASKELL AUTOMATED ASSESSMENT PLATFORM

This section presents our in-house AAP (named *Haskell Automated Assessment Platform – HAAP*), developed over the last 5 years to automatise the process of collecting and processing students' ongoing progress. It provides *continuous feedback* to the students via a website, and so-called *milestone feedback* after each of the two phases. The latter consists of a more detailed report to be used by the instructors for grading, and partly made available to students. To cater to our growing needs and amortise our yearly setup costs, HAAP has evolved from assignment-specific collections of scripts to an extensible, generic Haskell platform that can be instantiated for each assignment. We start by describing how HAAP is structured and how it can be reused (Section 4.1), followed by a detailed presentation of one instantiation based on the one for the PLab unit in 17/18 (Section 4.2).

Development. The homepage for HAAP development can be found at https://github.com/haslab/HAAP. It includes the complete source code for the generic HAAP platform, a series of examples – ranging from the minimalistic example from Section 4.1 to a mock-up of our instantiation for the PLab unit presented in Section 4.2 – and detailed installation and deployment instructions.

[7] Until 16/17, students were asked at this point to apply their newly acquired knowledge on inductive data types.

4.1 A Generic Platform

HAAP is developed in Haskell and builds on several libraries and tools available on Hackage.[8] It is modelled as the analogue of a batch script that assesses a project assignment and generates feedback in various forms.

In the style of popular Haskell frameworks such as Hakyll or XMonad, a HAAP script starts from an initial configuration that amounts to a global description of the project as a Haskell record:

```
data Project = Project { ... }
```

that includes the project's name and its filesystem path, together with listings of student groups and tasks. Each task is comprised by a set of files that can have both local (templates or oracles used for assessment) and/or remote (student solutions) representations.

Scripting is done in the **Haap** monad, that provides basic logging and error-handling capabilities:

```
data Haap plugins m a = ...
instance ... ⇒ Monad (Haap plugins m)
```

The default base monad `m` for a **Haap** script is **IO**, to enable basic interaction with the operating system. Other base monads may offer different functionality, such as **Sh** for shelly scripting or **Async** for async concurrency. For extensibility, everything else comes in the form of `plugins`. Following the design of web frameworks such as Yesod and Happstack, HAAP plugins are encoded as monad transformers.

To allow plugins to be combined, loaded plugins form a monad transformers stack. We can run a **Haap** script with the empty stack, given by the **IdentityT** monad transformer:

```
runHaap :: Project → Haap IdentityT IO a → IO a
```

Plugins can be stacked using monad transformer composition:

```
newtype ⊙ t1 t2 m a = ⊙ (t1 (t2 m a))
```

A plugin `p` is referenced via a unique type and has initialization arguments **PluginI** `p`, a monad transformer **PluginT** `p`, and a **usePlugin** function that loads the plugin onto the top of the stack:

```
class MonadTrans (PluginT p) ⇒ HaapPlugin p where
  type PluginI p = (r :: ∗) | r → p
  type PluginT p = (t :: (∗ → ∗) → (∗ → ∗)) | t → p
  usePlugin :: Haap plugins m (PluginI p)
            → Haap (PluginT p ⊙ plugins) m a → Haap plugins m a
```

We also need the ability to lift a plugin operation to the plugins stack:

```
class HasPlugin p plugins where
  liftPlugin :: Monad m ⇒ PluginT p m a → plugins m a
```

Writing a new plugin amounts to a new named type `p`, an instance of **HaapPlugin** `p` and an instance of **HasPlugin** `p` (**PluginT** `p` ⊙ `plugins`). When it is useful to extend the base **Haap** monad with additional functionality, e.g., to execute a shelly script, we provide functions such as:

```
sh :: MonadTransControl plugins ⇒ Haap plugins Sh a → Haap plugins IO a
```

that wraps the `shelly :: Sh a → IO a` function inside a **Haap** script. This is in fact a general pattern frequently used in Yesod and Happstack. Operationally, it requires the ability to disassemble the plugin stack, run `shelly` on the base monad, and reassemble the plugin stack back as it was. To play this trick the loaded `plugins` must, in Haskell jargon, be instances of **MonadTransControl**.

[8] https://hackage.haskell.org

```
main = runHaap defProject $ usePlugin defHakyllArgs $ do
  usePlugin defHspecArgs $ renderHspec "ord.html" $
    bounded "Int" [97,98,3,4] $ λx →
    bounded "Char" "abcd" $ λy →
    testEqual x (ord y)
  hakyllRules $ create ["index.md"] $ do
    route (setExtension "html")
    compile $
      makeItem "#Ord…[spec](ord.html)" >>= renderPandoc
```

Fig. 2. Minimalistic HAAP example.

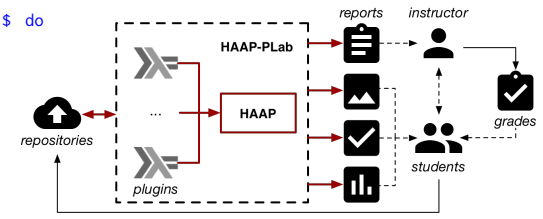


Fig. 3. Overview of HAAP-PLab.

The plugin abstraction also allows grouping plugins into classes. For instance, HAAP supports different internal database implementations via the interface:

```
class HaapPlugin db ⇒ HaapDB db where
  type DBQuery db a = (r :: ∗) | r → db a
  type DBUpdate db a = (r :: ∗) | r → db a
  queryDB :: (HasPlugin db plugins) ⇒ DBQuery db a → Haap plugins m a
  updateDB :: (HasPlugin db plugins) ⇒ DBUpdate db a → Haap plugins m a
```

A minimalistic example of HAAP in action is given in Fig. 2. It uses the Hspec plugin to write a specification that tests `ord`-equality of all combinations of given **`Int`**s and **`Char`**s. The `renderHspec` operation relies on the Hakyll plugin to generate a webpage (`ord.html`) with the test results, and requires the Hakyll plugin to be loaded onto the stack. It finishes by using Hakyll's built-in pandoc integration to generate a front webpage (`index.html`) from a markdown description.

Some of the most relevant HAAP plugins currently available are:

Sources Abstract plugin for manipulating version-control student repositories, and a concrete SVN plugin.

Databases Abstract plugin for managing internal state, and concrete acid-state and non-ACID binary serialization plugins.

Websites Hakyll plugin to automate static website generation.

Testing Hspec plugin that provides a test specification language combining HUnit and QuickCheck tests, and operations for running specifications and rendering results as HTML tables.

Documentation Haddock plugin that combines the Haddock tool and the Haddock-library, haskell-src-exts and syb libraries to generate HTML documentation and extract fine-grained documentation reports.

Code Analysis Standalone plugins for the Sourcegraph (analysis), hpc (coverage), Hlint (refactoring suggestions) and Homplexity (quality suggestions) tools. An additional code metrics plugin resorting to the haskell-src-exts and syb libraries.

Graphics Plugins for visualization and animation of Gloss code as webpages (combining the GHCJS Haskell to JavaScript compiler and the CodeWorld environment) and videos (combining OpenGL and FFmpeg).

General Plugins to automatise the setup and deployment of rankings and tournaments.

For security reasons, HAAP allows compiling student's code using *Safe Haskell* [Terei et al. 2012], a language extension that syntactically ensures that functions respect their tight type contracts with no overhead, and executing it inside *Linux containers*[9] with limited permissions and resources to protect against sophisticated attacks.

[9] https://linuxcontainers.org

4.2 A Concrete Instance

Each year HAAP is instantiated into a concrete AAP. This section presents HAAP-PLab, that essentially represents our 17/18 instantiation, and whose overall architecture is presented in Fig. 3. Students' ongoing work is automatically fetched from their SVN repositories, compiled and executed to generate feedback. The rest of this section explains in detail the provided feedback.

4.2.1 Continuous Feedback. HAAP-PLab periodically (on an hourly basis) generates a personal feedback page per group, and global rankings for competitive tasks. This avoids making the students cope with incomplete feedback and wait for long periods for the (manual) feedback from the instructors, as was the case before this reorganization. This feedback is made available through a web interface generated by the Hakyll plugin. A snapshot of this interface, populated with sample data, is available online.[10]

Task-level. Experiences in early years with the Mooshak system relied on instructor-defined test cases, blinded to the students, which raised barriers in understanding and fixing mistakes. HAAP-PLab requires students to provide their own test-scenarios for each of the tasks, which are used to compare their solutions with an oracle developed by the instructors. For each non-open task, HAAP-PLab provides task-level detailed reports on the correctness of the students' solutions (using the Hspec plugin, Fig. 4a) and the coverage of their test-scenarios (using the hpc plugin). Such reports are also connected to a Gloss GUI developed by the instructors where the oracle behaviour can be inspected for the students' tests, relying on the CodeWorld plugin (Fig. 4b).

Group-level. Besides task-specific feedback, the students are provided with group-level information regarding the overall quality of their code base. This process is based on the Haddock, Homplexity and Hlint plugins to automatically provide hints and recommendations on-the-large (Fig. 4f). Students are encouraged to document and refactor their code as they learn new programming concepts, and this feedback is key to this task.

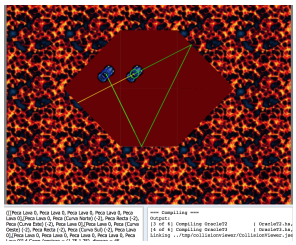
Global. Lastly, global feedback is provided regarding the open tasks, allowing students to compare their solutions among themselves. In Task 3, HAAP-PLab uses the Hspec plugin test the students' submissions for fixed instructor-defined test-scenarios, which are then passed to the ranking plugin to generate a leaderboard (Fig. 4c). To avoid biased solutions, these test-scenarios are different from the ones used for the final evaluation. In Task 6, HAAP-PLab uses the tournament plugin to randomly pair and compare student submissions and generate the corresponding bracket (Fig. 4d). Matches are animated using a Gloss GUI developed by the instructors using the CodeWorld plugin, or made persistent through the video generation plugin (Fig. 4e). This allows students to quickly verify the effectiveness of their strategies.

4.2.2 Milestone Feedback. After each phase, the students' submissions are collected and evaluated. Collection is done by taking a snapshot of the student's repositories and evaluation by measuring a list of criteria, listed below, and calibrating what instructors consider to be reasonable values through manual inspection of a handful of cases. HAAP-PLab generates a CSV report with this data that is later analysed using standard spreadsheet tools. A summary of these results is sent back to the students, together with their evaluation for each task.

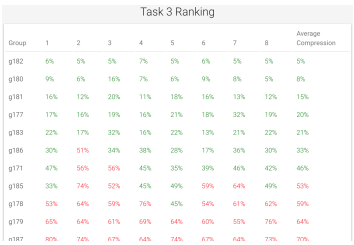
Task-level. All tasks except for Task 5 are evaluated using unit testing for *correctness* using the Hspec plugin. To ensure a *complete*, *uniform* and *deterministic grading*, HAAP-PLab runs the testing framework with a pre-defined and fixed set of tests and randomness seeds developed by the instructors. Task 3 is evaluated by measuring the average rank of the students' submission for

[10] https://haslab.github.io/HAAP/examples/plab/site/

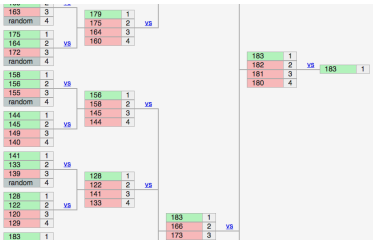
(a) Correctness feedback (T2 17/18).



(b) Visual feedback (T3 17/18).

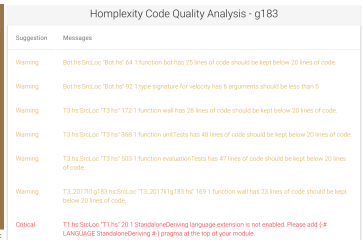


(c) Ranking feedback (T3 16/17).



(d) Tournament feedback (T6 17/18).

(e) Visual feedback (T6 16/17).



(f) Code quality feedback (17/18).

Fig. 4. Snapshots of different feedback provided to the students by HAAP-PLab.

all tests. Task 6 is evaluated by playing the students' bot against a fixed set of instructor-designed strategies. To make assessment as transparent as possible, the instructor tests and feedback results are provided to students through the same web interface used for continuous feedback.

There exist a few techniques for analysing *test effectiveness* in Haskell [Cheng et al. 2016]. HAAP-PLab uses *mutation testing* at task-level: the instructors design a set of faulty variations of the oracle, known as *mutants*, and the HAAP-PLab checks how many of the mutants' erroneous behaviours are detected by the student's test-scenarios.[11] HAAP-PLab also measures the *coverage* of student test-scenarios against the instructor oracles using the hpc plugin (this differs from the continuous feedback provided to the students that tests the coverage for their own code).

Group-level. Measuring the quality of the students' code for all 6 tasks can be subjective. To automatise this process and evaluate all assignments consistently, as in continuous feedback, HAAP-PLab provides the instructors with group-level information compiling several metrics, such as cyclomatic complexity (with the Sourcegraph plugin), the number of user-defined data types and of higher-order functions used, the average length of declarations (with the code metrics plugin), and the size, coverage, and (syntactic) richness of the code documentation (with the Haddock plugin). In turn, the instructors calibrate the scores for code and documentation quality by manually inspecting a range of student assignments.

Individual-level. Part of the evaluation still requires a personal interaction with individual students. After submitting the 6 tasks, each group must present their work to their instructor in an *oral defence*, justifying their decisions and explaining used algorithms. Task 5 is evaluated during this defence, since it consisted of a GUI development, and not automatically evaluated. Furthermore, different students from the same group are distinguished in the defence, also based on the contribution logs from the SVN plugin; and an open discussion with the students allows

[11] We could use the MuCheck [Le et al. 2014] library to automatically generate mutants, but refrain to do so because some of the generated mutations could correspond to specific corner cases or even violate the game's rules, and would provide an unfair evaluation criterion. We find it preferable for instructors to define the more relevant mutations for a particular game.

the identification of possible adjustments to their final grades. Finally, the final grade takes into account the perceived quality of the reports by their instructor, and indications of plagiarism by the Moss service [Aiken 1994] lead to failure of the affected students.

5 LESSONS LEARNED

Often in the paper we have reaffirmed the lessons that we have learned as instructors of PLab for the last 5 years, and these have motivated improvements to our teaching methodology. In general, we feel that, despite the initial overhead of developing HAAP, gamification and continuous feedback has been beneficial for both students – promoting their engagement and helping overcome the initial lack of technical knowledge – and instructors – easing the evaluation and support of an increasingly large student body. Nonetheless, there are naturally still aspects we feel can be improved in future years.

Below we summarise more fine-grained perceptions and what we believe others can learn from this experience. We also make explicit how we perceive the success of the three challenges proposed in the introduction, namely (C1) targeting fresh and inexperienced students, (C2) over 200 students enrolled, and (C3) high drop-out rates.

What we learned as instructors. *Targeting grid-based arcade games* keeps the students more involved (C1,C3). *Continuous events (time, moving obstacles, etc.) are hard* to get right and should be delayed to later tasks (C3). *Open, competitive and incremental tasks* stimulate students to proactively investigate advanced algorithms (C1). *Too much feedback can be harmful* when students fail to postpone fixing small (and time consuming) errors. *Early usage of rich data types pays off*, by discouraging students (especially those with previous programming experience) from falling into less structured programming patterns that abuse string processing. *Tasks can provide more fine-grained feedback*, to promote early engagement and motivate students that fail to combine intermediate functions into a working solution (C1). *Manual assessment remains important*, since automatic assessment provides good indicators but does not always reflect students performance and is not effective to discern individual group-element contributions (C3). *Off-the-shelf analysis tools provide informal feedback* that fosters good (functional) programming practices, even though the reports are sometimes confusing for students (C2). *Web-based feedback helps raising student awareness* to qualitative aspects that do not directly reflect on grading (C2). *Visual feedback is great but not always effective*, as it requires the complete task behaviour to have an intuitive graphical illustration. *Fraud detection tools discourage copies* if used early in the project, and can be performed using a combined analysis of Moss and version-control logs.

What we learned as developers of HAAP. *Haskell testing frameworks are very powerful for TDD*, allowing to easily provide simple and slightly different counter-examples (C1,C2). *Random testing can mitigate floating-point errors* by favouring test cases that minimize the difference between the oracle and the tested solution (C2). *Scripting in Haskell can be troublesome* due to Haskell's interplay of laziness, exceptions and IO, requiring the use of strictness annotations to prevent uncaught errors and exhausted server resources. *Measuring quality of functional code is difficult*, and there is space for more intuitive metrics and more robust tools, for instance, by plugging to the GHC API. *Static generation of feedback reports* on a periodic basis can be computationally heavy on the AAP and delay feedback loops (C2). *JavaScript visualization is mature*, combining GHCJS and CodeWorld, provides a lightweight client-side alternative to running animations on the AAP server; improved Gloss integration allowed us to smoothly compile student solutions into an animated hall of fame (C1,C2,C3).

What we perceived from students. At the end of the semester, students are required to fill feedback forms, evaluating their perceived quality of the unit and allowing them to comment on positive and negative aspects. The university compiles, anonymises and makes this feedback available to the instructors. Feedback has been overall positive, praising the appeal of developing an interactive game, the challenges proposed, the availability of the instructors, and what students learned by working in teams. As students put it, *"this unit was very demanding, but still very appealing"* (16/17) in *"developing a spirit group and enthusiasm for programming"* (17/18), *"(...) and helped us learning to solve problems and optimize solutions"* (16/17).

Complaints regarded mostly technical and curricular aspects, such as *"the feedback system needs improvement (...) and takes too long"* (17/18), around 2 hours, or *"the work load in comparison to the ECTS credits is too high (...)"* (17/18). Some students suggested that more incentives could be explored to frighten away inexperience fears, especially in initial stages: *"(...) it would be worth to consider developing various smaller and simpler projects throughout the semester to gain experience"* (16/17). A 17/18 comment reflects our feeling about the current state of PLab: *"the complexity of this project forced me to work and learn in an area where my knowledge was short, ending up accomplishing results that I had no idea I could"*. We believe that our unit restructuring kept students motivated to work actively on their project and contributed to set them up for a successful CSE experience, hopefully mitigating the drop-out rates. The feedback also includes overall unit statistics, in which we have observed an increase in student approval in both the PLab and FPro units over the years.

6 CONCLUSIONS

This paper reported on our experience of redesigning an introductory FP course unit at the University of Minho, based on programming a complete grid-based interactive arcade game, and building an AAP – HAAP– that provides proper guidance and rigorous grading, all made possible due to the tight FP-powered integration between the assignment model and the AAP. HAAP is publicly available and can be easily instantiated to assignments that fit the proposed project model. This reorganisation required more investment from instructors, namely to support the development of the system and the design of a project model amenable for automatic assessment, and students have been asked for greater dedication to design increasingly more advanced games.

Ultimately, we have found this experience to be fruitful for both instructors and students, and more successful in the teaching of FP concepts and software development practices. In the future, we expect to address the sometimes long feedback loops, by supporting dynamic webpages in HAAP that only generate feedback on demand. We also plan to improve students motivation in the initial tasks of each phase of the project by providing a more explicit notion of progress via microtasks or more immediate rewards. For this purpose, we envision extending HAAP with additional gamification plugins.

ACKNOWLEDGMENTS

The authors would like to thank the precursors of the 20-year functional programming culture and FPro unit at our university, and all the instructors and TAs that have been involved in the PLab unit throughout the years. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation – COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia as part of project UID/EEA/50014/2013.

REFERENCES

Peter Achten. 2008. Teaching functional programming with soccer-fun. In *Proceedings of the 2008 international workshop on Functional and declarative programming in education, FDPE@ICFP 2008, Victoria, BC, Canada, September 20 - 28, 2008.* ACM, 61–72. https://doi.org/10.1145/1411260.1411270

Alex Aiken. 1994. Moss: A System for Detecting Software Similarity. http://theory.stanford.edu/~aiken/moss/. (1994). Accessed: 2018-03-14.

Tiffany Barnes, Eve Powell, Amanda Chaffin, and Heather Richter Lipford. 2008. Game2Learn: improving the motivation of CS1 students. In *Proceedings of the 3rd International Conference on Game Development in Computer Science Education, GDCSE 2008, Miami, FL, USA, February 28 - March 3, 2008.* ACM, 1–5. https://doi.org/10.1145/1463673.1463674

Jessica D. Bayliss and Sean Strout. 2006. Games as a "flavor" of CS1. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2006, Houston, Texas, USA, March 3-5, 2006.* ACM, 500–504. https://doi.org/10.1145/1121341.1121498

Susan Bergin and Ronan Reilly. 2005. The influence of motivation and comfort-level on learning to program. In *Proceedings of the 26th Annual WorkShop of the Psychology of Programming Interest Group*, Vol. 17. PPIG, 293–304.

Joachim Breitner and Chris Smith. 2017. Lock-step simulation is child's play (experience report). *PACMPL* 1, ICFP (2017), 3:1–3:15. https://doi.org/10.1145/3110247

Manuel M. T. Chakravarty and Gabriele Keller. 2004. The risks and benefits of teaching purely functional programming in first year. *J. Funct. Program.* 14, 1 (2004), 113–123. https://doi.org/10.1017/S0956796803004805

Carl Chang, Peter J. Denning, James H. Cross II, Gerald Engel, Robert Sloan, Doris Carver, Richard Eckhouse, Willis King, Francis Lau, Susan Mengel, Pradip Srimani, Eric Roberts, Russell Shackelford, Richard Austing, C. Fay Cover, Gordon Davies, Andrew McGettrick, G. Michael Schneider, and Ursula Wolz. 2001. *CC2001: Computing Curricula 2001 Computer Science.* Technical Report. IEEE & ACM.

Yufeng Cheng, Meng Wang, Yingfei Xiong, Dan Hao, and Lu Zhang. 2016. Empirical Evaluation of Test Coverage for Functional Programs. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016.* IEEE Computer Society, 255–265. https://doi.org/10.1109/ICST.2016.8

Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa arcade. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2003, Uppsala, Sweden, August 28, 2003.* ACM, 7–18. https://doi.org/10.1145/871895.871897

Marcus Crestani and Michael Sperber. 2010. Experience report: growing programming languages for beginning students. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010.* ACM, 229–234. https://doi.org/10.1145/1863543.1863576

Stephen H. Edwards. 2003a. Improving student performance by evaluating how well students test their own programs. *ACM Journal of Educational Resources in Computing* 3, 3 (2003), 1:1–1:24. https://doi.org/10.1145/1029994.1029995

Stephen H. Edwards. 2003b. Rethinking computer science education from a test-first perspective. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA.* ACM, 148–155. https://doi.org/10.1145/949344.949390

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2009. A functional I/O system or, fun for freshman kids. In *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009.* ACM, 47–58. https://doi.org/10.1145/1596550.1596561

Patricia Haden. 2006. The incredible rainbow spitting chicken: teaching traditional programming skills through games programming. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52.* Australian Computer Society, Inc., 81–89.

HaskellWiki. 2018. Haskell as a First Language. https://wiki.haskell.org/Haskell_in_education#Haskell_as_a_first_language. (2018). Accessed: 2018-03-12.

Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *10th Koli Calling International Conference on Computing Education Research, Koli Calling '10, Koli, Finland, October 28-31, 2010.* ACM, 86–93. https://doi.org/10.1145/1930464.1930480

John Impagliazzo, Susan Conry, Joseph L.A. Hughes, Liu Weidong, Lu Junlin, Andrew McGettrick, Victor Nelson, Eric Durant, Herman Lam, Robert Reese, and Lorraine Herger. 2016. *CE2016: Computer Engineering Curricula 2016.* Technical Report. ACM & IEEE.

Alexandru Iosup and Dick H. J. Epema. 2014. An experience report on using gamification in technical higher education. In *The 45th ACM Technical Symposium on Computer Science Education, SIGCSE '14, Atlanta, GA, USA - March 05 - 08, 2014.* ACM, 27–32. https://doi.org/10.1145/2538862.2538899

Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. MuCheck: an extensible tool for mutation testing of haskell programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014.* ACM, 429–432. https://doi.org/10.1145/2610384.2628052

José Paulo Leal and Fernando M. A. Silva. 2003. Mooshak: a Web-based multi-site programming contest system. *Softw., Pract. Exper.* 33, 6 (2003), 567–581. https://doi.org/10.1002/spe.522

Frederick W. B. Li and Christopher Watson. 2011. Game-based concept visualization for learning programming. In *Proceedings of the 3rd international ACM workshop on Multimedia technologies for distance learning*. ACM, 37–42. https://doi.org/10.1145/2072598.2072607

Ben Lippmeier. 2010. Gloss: Painless 2D vector graphics, animations and simulations. http://gloss.ouroborus.net. (2010). Accessed: 2017-02-18.

Christoph Lüth. 2003. Haskell in Space. *J. Funct. Program.* 13, 6 (2003), 1077–1085. https://doi.org/10.1017/S0956796803004891

Marco T. Morazán. 2010. Functional Video Games in the CS1 Classroom. In *Trends in Functional Programming - 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers (LNCS)*, Vol. 6546. Springer, 166–183. https://doi.org/10.1007/978-3-642-22941-1_11

Nachiappan Nagappan, Laurie A. Williams, Miriam Ferzli, Eric Wiebe, Kai Yang, Carol Miller, and Suzanne Balik. 2003. Improving the CS1 experience with pair programming. (2003), 359–362. https://doi.org/10.1145/611892.612006

Ilias O. Pappas, Michail N. Giannakos, and Letizia Jaccheri. 2016. Investigating Factors Influencing Students' Intention to Dropout Computer Science Studies. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2016, Arequipa, Peru, July 9-13, 2016*. ACM, 198–203. https://doi.org/10.1145/2899415.2899455

Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth S. Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *SIGCSE Bulletin* 39, 4 (2007), 204–223. https://doi.org/10.1145/1345375.1345441

Kris Powers, Stacey Ecott, and Leanne M. Hirshfield. 2007. Through the looking glass: teaching CS0 with Alice. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2007, Covington, Kentucky, USA, March 7-11, 2007*. ACM, 213–217. https://doi.org/10.1145/1227310.1227386

Ilya Sergey. 2016. Experience report: growing and shrinking polygons for random testing of computational geometry algorithms. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. ACM, 193–199. https://doi.org/10.1145/2951913.2951927

Kelvin Sung, Michael Panitz, Scott A. Wallace, Ruth Anderson, and John Nordlinger. 2008. Game-themed programming assignments: the faculty perspective. (2008), 300–304. https://doi.org/10.1145/1352135.1352241

David Terei, Simon Marlow, Simon L. Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*. ACM, 137–148. https://doi.org/10.1145/2364506.2364524

Ville Tirronen, Samuel Uusi-Mäkelä, and Ville Isomöttönen. 2015. Understanding beginners' mistakes with Haskell. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000179

Arto Vihavainen, Matti Paksula, and Matti Luukkainen. 2011. Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42nd ACM technical symposium on Computer science education, SIGCSE 2011, Dallas, TX, USA, March 9-12, 2011*. ACM, 93–98. https://doi.org/10.1145/1953163.1953196

Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students' learning using test my code. In *Innovation and Technology in Computer Science Education conference 2013, ITiCSE '13, Canterbury, United Kingdom - July 01 - 03, 2013*. ACM, 117–122. https://doi.org/10.1145/2462476.2462501