The effect of a web-based coding tool with automatic feedback on students' performance and perceptions

Luciana Benotti
Universidad Nacional de Cordoba / CONICET
Cordoba, Argentina
benotti@famaf.unc.edu.ar

Federico Aloi
Universidad Nacional de Quilmes
Buenos Aires, Argentina
federico.aloi@unq.edu.ar

Franco Bulgarelli
Mumuki Project
Buenos Aires, Argentina
franco@mumuki.org

Marcos J. Gomez
Universidad Nacional de Cordoba
Cordoba, Argentina
mgomez4@famaf.unc.edu.ar

ABSTRACT

In this paper we do three things. First, we describe a web-based coding tool that is open-source, publicly available and provides formative feedback and assessment. Second, we compare several metrics on student performance in courses that use the tool versus courses that do not use it when learning to program in Haskell. We find that the dropout rates are significantly lower in those courses that use the tool at two different universities. Finally we apply the technology acceptance model to analyse students perceptions.

CCS CONCEPTS

• **Language Classifications → Applicative (functional) languages**; • **Computers and Education → Computer-assisted instruction**;

KEYWORDS

Functional programming; Haskell; Coding tools; Replication; Automatic assessment

ACM Reference format:
Luciana Benotti, Federico Aloi, Franco Bulgarelli, and Marcos J. Gomez. 2018. The effect of a web-based coding tool with automatic feedback on students' performance and perceptions. In *Proceedings of SIGCSE '18: The 49th ACM Technical Symposium on Computing Science Education, Baltimore , MD, USA, February 21–24, 2018 (SIGCSE '18),* 6 pages.
https://doi.org/10.1145/3159450.3159579

1 INTRODUCTION

While many tools are developed in order to support the teaching and learning of programming, few such tools ever achieve widespread adoption and use. A survey [2] found that one of the most common reasons is the teachers' unwillingness to adopt a new system until its usefulness has been demonstrated.

In general, the evaluations of such tools use ad-hoc metrics and are conducted on a single course (for notable exceptions see [8]). As argued in [8], there is a critical need for replicating studies in different contexts and for evaluating various contributing factors. It is also necessary to use standard metrics in order to better understand the reasons why certain results occur. In this paper we hypothesize that coding tools may have an effect on the high dropout rates observed in introductory programming courses [15].

The main contributions of this paper are:

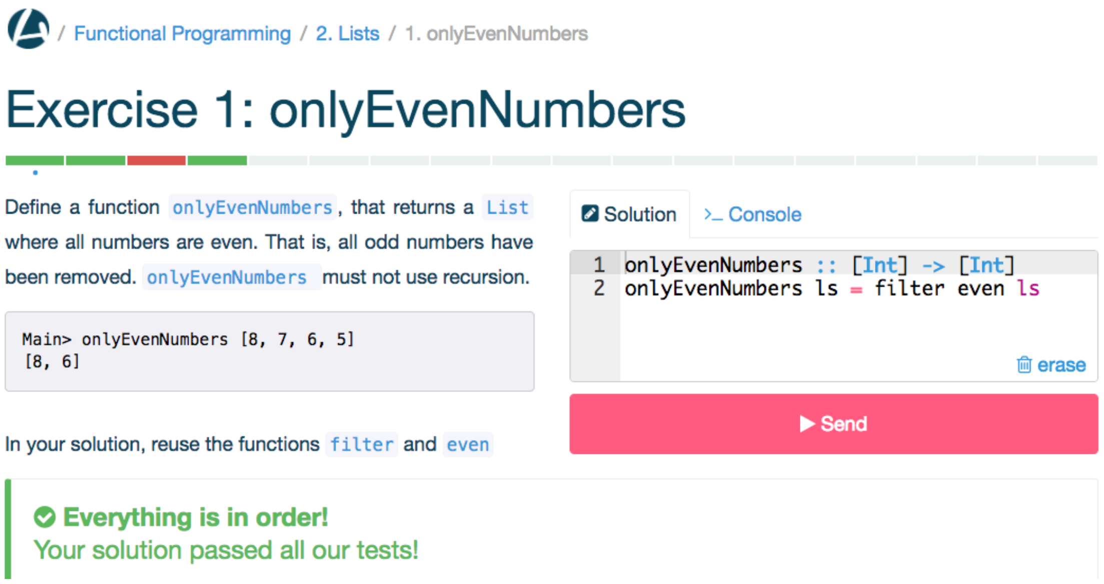
- We describe a web-based coding tool that is open-source and provides formative feedback and assessment.
- We compare several metrics on student performance in courses that use the tool versus courses that do not use it when learning to program in Haskell.
- We find that the dropout rates are significantly lower in those courses that use the tool at two different universities.
- We apply a standard model, the technology acceptance model (TAM) to analyse students perceptions.

According to a recent review [9], less than 10% of the educational programming tools support functional programming. This paper focuses on the language Haskell, a traditional functional programming language that has gained popularity in recent years.

We begin the paper by reviewing previous work on educational coding tools for functional programming and on previous studies on the effect of such tools on students' performance and perceptions. Then, we describe the online coding tool that we use, called Mumuki, and its rationale. A screenshot can be seen in Figure 1. Moreover, we address the study design followed by our findings. We analyse student passing, failure and dropout rates. Finally, we discuss our findings when applying the technology acceptance model on the courses.

2 PREVIOUS WORK

The topic of online coding tools has received plenty of attention during the last years. Well known examples are Codingbat [13], CodeRunner [12], CodeLab [1] and CloudCoder [14]. These online tools have been shown to be useful helping students' master the syntax of imperative programming languages such as Java and Python [8]. In spite of the attention this topic has received from researchers, the use of online coding tools is not widespread in education [2]. There are studies [8] that use these tools in order to identify students at-risk of dropout. However, longitudinal studies,



Exercise 1: onlyEvenNumbers

Define a function `onlyEvenNumbers`, that returns a `List` where all numbers are even. That is, all odd numbers have been removed. `onlyEvenNumbers` must not use recursion.

```
Main> onlyEvenNumbers [8, 7, 6, 5]
[8, 6]
```

In your solution, reuse the functions `filter` and `even`

```
onlyEvenNumbers :: [Int] -> [Int]
onlyEvenNumbers ls = filter even ls
```

Everything is in order!
Your solution passed all our tests!

Figure 1: Mumuki screenshot. The left hand panel shows an exercise description. The right hand panel shows the editor where a student wrote a solution. Finally, the bottom of the figure shows the feedback provided when the solution is correct.

where the effect of an intervention would be measured across a number of courses or institutions, are rare.

This paper provides evidence that this kind of systems are useful in introductory programming courses at two different institutions. In particular, we show their usefulness with functional programming languages. In this section we review previous work on educational coding tools for functional programming. As we mentioned in Section 1, almost all educational coding tools focus on imperative languages. To the best of our knowledge, in addition to ours there is only one online coding tool for Haskell that was developed with educational purposes: Ask-Elle.

Ask-Elle [6, 7] is a web-based tutor that shares some features with Mumuki. As well as Mumuki it presents exercises in Haskell and generates automatic feedback to students' programs. Also, similarly to Mumuki, it evaluates the students' solutions using test cases.

Differently from Mumuki, Ask-Elle was conceived as a tutor that provides hints during the intermediate steps of programming a solution to an exercise. Ask-Elle [6] tries to predict which basic steps have to be taken in order to arrive at a solution. They call these steps a "programming strategy" and they derive them automatically from model solutions provided by the teachers. However, in their experiments, they find that it is very difficult to predict the size of the step that students take. Furthermore, often a student's solution cannot be matched to one of the model solutions provided by the teachers by their program transformations. As a result they cannot provide feedback for almost 40% of the student's submissions. The authors note that students usually take steps that are bigger than those expected by the system, frequently submitting complete solutions in one step. Mumuki expects a complete solution whenever it receives a submission. Therefore it is able to provide feedback for all the submissions received.

Another difference between the two tools is that Ask-Elle requires teachers to define model solutions to the exercises while Mumuki does not. Ask-Elle uses QuickCheck [4] which generates random tests cases. QuickCheck developers acknowledge that the major limitation of the software is that there is no measurement of test coverage, since the test cases are generated at random. Mumuki require teachers to manually define test cases so that special attention can be put into border cases and problematic branches. Unlike Ask-Elle, Mumuki also provides a Haskell console for each exercise where students can try their own test cases. Furthermore, in order to verify solution quality, Mumuki allows teachers to define expectations on the submissions. We explain this novel feature in the next section.

Ask-Elle was evaluated at a single university, the Utrecht University, by 40 students. The students used the tool during two laboratory sessions whose duration is not reported. The students attitude towards the tool was also evaluated through a questionnaire as we did. However, the authors do not report following any standard methodology. The students rated the tool an average of 2.82 in the question "The step-size of the tutor corresponded to my intuition" on a Likert scale from 1 to 5. The effect on course dropouts and course pass and failure rates was not reported.

Kumar [10] argues that online coding tools can be used to increase the self-confidence of students in Computer Science. This is particularly useful in the first CS courses when student's self-confidence is lower. Kumar also claims that the impact is stronger on female students. However, they do not report the effect on course performance metrics as we do.

To the best of our knowledge there is only one study [1] where the authors successfully applied a web-based coding tool, CodeLab, across two different institutions. The results and the discussion are encouraging. However, the comparison is purely qualitative, the impact of the tool over quantitative metrics, such as failure and pass rates is not reported. Grades are reported, but only for one of the institutions. The effect on students' performance and perceptions of interactive books [5] and visualisation tools [11] has been better evaluated than the effect of coding tools.

3 MUMUKI: AN ONLINE CODING TOOL

Mumuki is an open-source[1] web-based coding tool that provides automatic formative feedback and assessment, developed in Argentina. This platform provides assistance for teachers and students, in a process conducted by practice where theory arises from the exercises. It supports 17 programming languages, including Haskell, Prolog, Python, JavaScript, C and Ruby. Mumuki is currently used at eight Argentinian universities, five high schools and two coding clubs, with more than 1500 active users per month.

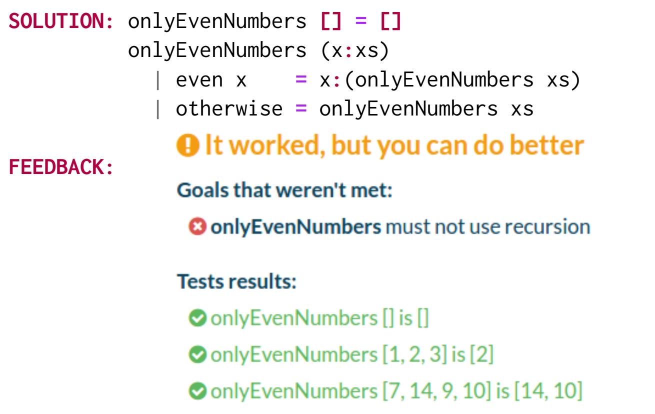
A screenshot can be seen in Figure 1. Practice within the platform is organised across topics with many programming exercises[2] each. The interface includes a progress bar that shows which exercises have been solved (in green), which have been attempted but are incorrect (in red or yellow) and which have not been attempted yet (in grey). The current exercise is marked with a blue dot. The students can solve the exercises in any order they want, but Mumuki suggests an order with the progress bar.

From the student point of view, an exercise includes the description of the problem with at least one example and a list of functions that can or must be reused, as can be seen in the figure in the left hand panel. In the right hand panel it includes a tab with an editor for the solution and another tab with an interactive console where the solution and the reusable functions can be tested. The automatic feedback for the student's solution is shown at the bottom of the screen once she presses the "Send" button . In Figure 1, the solution passed all the tests.

From the teacher point of view, an exercise includes two parts. On the one hand, it includes a description of the problem the student should solve by providing a program, in which generally, no more than a few a few lines of code are required. On the other hand, the exercise designer provides a way to automatically compute feedback for the solution of the exercise at two different levels: correctness and quality. For correctness, a suite of representative unit tests are provided by the teacher and executed against the final program. The final program could include auxiliary functions programmed by the teacher. Although correctness cannot be assured by this technique, it proved to be good enough for the complexity of the existent exercises. Regarding the quality of the student's solution, an analysis of the abstract syntax tree is performed. By using an editor tool, the teacher can pick several predefined patterns, called *expectations* in Mumuki. Expectations are defined so that they must be present or not present in the student's solution. When the student submits a solution, each expectation is evaluated against it.

Student solutions are assessed by Mumuki with three possible values: red, yellow or green. Red indicates that some test result is not correct. Yellow means that all tests are correct but some expectation is not met. An exercise is marked green when all tests are correct

[1]Source code available at https://github.com/mumuki.
[2]The exercises are available at https://mumuki.io (in Spanish)



```
SOLUTION: onlyEvenNumbers [] = []
          onlyEvenNumbers (x:xs)
            | even x      = x:(onlyEvenNumbers xs)
            | otherwise = onlyEvenNumbers xs
```

FEEDBACK:

It worked, but you can do better

Goals that weren't met:

onlyEvenNumbers must not use recursion

Tests results:

onlyEvenNumbers [] is []

onlyEvenNumbers [1, 2, 3] is [2]

onlyEvenNumbers [7, 14, 9, 10] is [14, 10]

Figure 2: Sample solution to an exercise which is assessed as yellow by Mumuki because some expectation is not met.

and all expectations are met. The figure shows an exercise assessed as green. Below we exemplify the other two colors.

The exercise in Figure 1 asks the student to define a function called `onlyEvenNumbers` that returns only the even numbers of a given list. The exercise description is illustrated by the example `onlyEvenNumbers [8,7,6,5] = [8,6]`. The following unit tests are provided by the teacher and not shown to the student until she submits a solution:

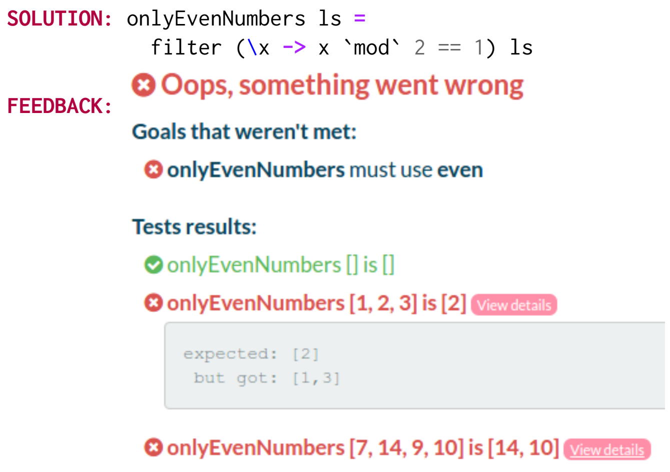
- `onlyEvenNumbers [] = []`.
- `onlyEvenNumbers [1,2,3] = [2]`.
- `onlyEvenNumbers [7,14,9,10] = [14,10]`.

The teacher also defines the following expectations. Because the use of `filter` is expected, direct recursion is not allowed. This is specified by the teacher as a pattern that *should not* be found in the student solution. Also, the teacher wants to encourage code reuse, so the solution must use the `even` function provided by Haskell's standard libraries. This is specified by the teacher as a pattern that *should* be found in the student solution.

In Figure 2 the student uses `even` but she uses direct recursion instead of using `filter` in the solution. Mumuki informs the test results are correct but explains that recursion must not be used. Therefore, the exercise is assessed by Mumuki as yellow: tests results are correct but some expectation is not met.

In Figure 3 the student did not use `even` and also made a mistake when programming her version of even. Instead of verifying that the modulus operation is zero, she compares it to one. Both the tests and one of the expectations fail, and this is reported by Mumuki. In this case, the exercise is assessed as red because some test results are incorrect.

Every solution submitted by a student is stored, thus the teacher has access not only to the final solution but also to all the steps that lead to it. The timestamp, the tests and expectations results are also stored, so the full history can be reproduced. It is also used to improve the exercises, to explore common mistakes, etc. The logged data is presented on a web page as shown in Figure 4 and can be accessed only by the teachers. In the web interface, the teachers can monitor students' progress. In the figure, the student has submitted ten different solutions for this exercise. The last one



```
SOLUTION: onlyEvenNumbers ls =
            filter (\x -> x `mod` 2 == 1) ls
```

FEEDBACK:

Oops, something went wrong

Goals that weren't met:

onlyEvenNumbers must use even

Tests results:

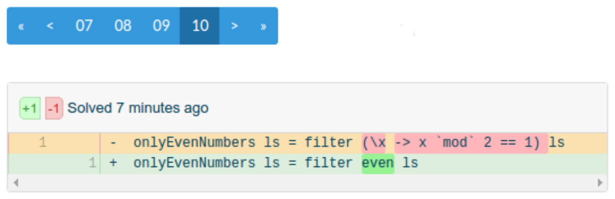
onlyEvenNumbers [] is []

onlyEvenNumbers [1, 2, 3] is [2] View details

```
expected: [2]
 but got: [1,3]
```

onlyEvenNumbers [7, 14, 9, 10] is [14, 10] View details

Figure 3: Sample solution to an exercise that is assessed as red by Mumuki because some tests results are incorrect.



+1 -1 Solved 7 minutes ago

```
- onlyEvenNumbers ls = filter (\x -> x `mod` 2 == 1) ls
+ onlyEvenNumbers ls = filter even ls
```

Figure 4: Two solutions for the same exercise and their difference are reported on the teacher's tool, using a diff highlighting tool.

is marked green by Mumuki since it passes all the tests and meets all the expectations.

4 STUDY DESIGN

In this section we describe the contexts where we conducted the observational studies in two different courses at two universities. We pose the following directional hypothesis to be tested at the two courses. H_0 *(null): There is no significant difference between the dropout rate at a course which uses Mumuki and at one that does not.* H_1 *(alt): The dropout rate at a course which uses Mumuki is significantly lower than that of a course that does not.*

4.1 Data Collection and Analysis

We performed interventions in two different public universities. In both of them we taught Haskell using Mumuki.

As we argued in Section 2, most previous work evaluating web-based coding tools are conducted within a single institution and a single course. We selected two institutions that introduce functional programming with Haskell. They have groups of students with different characteristics to replicate our observational study and validate our results. One of the public universities, called UTN, introduces Haskell to CS2 students while the other, called UNC, introduces Haskell to CS1 students. In total, 114 students participated in this observational study. 59 students at CS2 level and 55 students at CS1 level.

Students were asked to solve 82 programming exercises covering introductory functional concepts. Their understanding of the concepts was evaluated with a written test on paper. Neither Mumuki nor other compiler was used for the exam. The exam was manually corrected by the teachers. Students had two opportunities to pass the test; there was at least one month between them.

In order to evaluate their attitude towards Mumuki, we asked the students to complete a questionnaire designed following the Technology Acceptance Model (TAM) methodology [3].

4.2 The CS1 Intervention

UNC is a public university in Argentina where students are full time. Students attend courses approximately 35 hours per week. As a result, there are almost no students that study and work.

From August to October 2016 a university professor, two teaching assistants and one student assistant taught 15 four-hour functional programming lessons to CS1 students at UNC. The course was taught for eight hours a week divided in two days from 9am to 1pm. This is the first programming course of a 5-year degree in Computer Science. The course started with 55 students that enrolled and attended the course at least two days. From the 55 students, only 2 reported previous experience programming imperative languages and none of them reported experience with functional programming. The average age was 21 years old. Each four-hour lecture was divided in two stages of two hours each.

During the first stage, the students used Mumuki in the laboratory programming functions in Haskell. Mumuki provided automated formative feedback but also the professor and the teaching assistants walked around the laboratory answering questions about the exercises.

During the second stage, the students went to a classroom where the professor explained on the blackboard the concepts practiced in the laboratory, presented worked out examples and discussed the most commonly observed errors. No computer was used in the classroom. Students were encouraged to program in Haskell using paper and pencil and to propose test cases in order to manually evaluate their programs.

Summing up, during the course, students used Mumuki during 30 hours. They also used Mumuki outside the course but 85% of their submissions were made during class hours.

4.3 The CS2 Intervention

UTN is a public university in Argentina where students are part time. Students attend courses approximately 20 hours per week. As a result, students can study and work.

From March to May 2016 a university professor, two teaching assistants and six student assistants taught 9 four-hour functional programming lessons to CS2 students at UTN. The course was taught for four hours a week in a single day from 9am to 1pm. Before this course, all the students learn imperative programming in Pascal and C during CS1. The course started with 59 students that attended the course for at least two classes. In the previous semester, Pascal and C languages were used to teach classical control structures and basic data structures such as lists, stacks, and

	2015		2016		t-test	
	n	ratio	n	ratio	p	t
Passed Test	15	.33	22	.37	.62	.49
Passed Re-Test	3	.06	8	.14	.25	1.17
Dropout	13	.28	8*	.14	.05	2.01
Failed	15	.33	21	.35	.75	.32
Total	46	1	59	1	-	-

Table 1: Dropout, passed and failed quantities and ratios in CS2 at UTN in 2015 and 2016. Significant differences between 2015 and 2016 are marked with *, p and t values are reported for unpaired t-test results.

	2014		2015		2016		t-test	
	n	ratio	n	ratio	n	ratio	p	t
Passed Test	21	.32	16	.31	21	.38	.43	.79
Passed Re-Test	3	.05	4	.05	12*	.22	.02	2.3
Dropout	38	.58	34	.59	19*	.35	.01	2.6
Failed	4	.06	4	.05	3	.05	.75	.31
Total	66	1	58	1	55	1	-	-

Table 2: Dropout, passed and failed quantities and ratios in CS1 at UNC in 2014, 2015 and 2016. Significant differences between 2015 and 2016 are marked with *, p and t values are reported for unpaired t-test results.

queues. None of the students reported previous experience with functional programming. The average age was 22 years old.

During the classes, Mumuki was used to introduce new concepts and to practice previously learnt ones. When considered necessary by the professor or the teaching assistant, they stopped the practice and explained some concept in the blackboard or screen projector using Mumuki, discussing errors and sharing different solutions. These explanations lasted 10 minutes in average and never took more than 10% of the total time. Students were not asked to program on paper and they used Mumuki during the whole lecture.

Summing up, during the course, students used Mumuki during 36 hours. They also used Mumuki outside the course, 70% of their submissions were made during class hours.

5 FINDINGS

In this section we describe our findings. We first describe the passed, failed, and dropout ratios, in CS1 and CS2. Then we present our results of applying the Technology Acceptance Model (TAM).

5.1 Dropout, failure and pass metrics

Tables 1 and 2 show performance results for the year 2016, the only year Mumuki was used. Performance results for one or two previous years[3] are also reported. When Mumuki was not used, the students interacted directly with a compiler (Hugs or Ghci) in order to solve the programming exercises. The previous year courses only differ from the 2016 course in the coding tool used. The teaching staff, exercises, types of exam, etc, remained the same. The performance metrics are:

- Passed Test: Number (n) and ratio of students that passed the subject exam the first time they tried.
- Passed Re-Test: Number (n) and ratio of students that passed the subject exam the second time they tried. There is at least one month between the test and the re-test.
- Dropout: Number (n) and ratio of students that enrolled and attended at least two lectures. They either were absent or failed the first test. They were absent during the re-test.
- Failed: Number (n) and ratio of students that failed the test and the re-test.

In both tables, a statistically significant decrease in the dropout can be observed the year Mumuki was used. The statistical analysis used is unpaired t-tests, between the years 2015 and 2016 for all metrics. The values for p and T are reported in the tables. The dropout ratio at UTN in 2016 is .14 while it was .28 in 2015. The dropout ratio at UNC is .35 in 2016 while it was .59 in 2015. These results give evidence for rejecting the null hypothesis H_0(null) in favour of the alternative one H_1(alt): The dropout rate at a course which uses Mumuki is significantly lower than that of a course that does not. This hypothesis was tested in two contexts which are quite different. According to the results, the effect seems to be stronger on our CS1 sample but they are still significant in CS2.

[3]The year 2014 is not reported for UTN because the information was incomplete.

Table 2 shows that the difference between 2015 and 2016 is also significant for the metric Passed re-Test for UNC. We have observed that the students use Mumuki as a tool for preparing themselves for the re-test. During the time between the test and the re-test the course is already over and there are no regular lectures in order to prepare for the re-test. Mumuki gives them a chance to study at their own pace.

The dropout rate in our observational studies is high when comparing it to the one in other countries reported in previous work with an average of 20% [15]. Probable causes could be among the following. Both UNC and UTN are public universities and hence they are completely free. Some students work during their studies and others have weak backgrounds in mathematics.

5.2 Technology acceptance model

Finally we analyse students' perceptions regarding Mumuki's usefulness and ease of use with the Technology Acceptance Model (TAM) [3]. The survey was not mandatory and was completed by 87 students in 2016. We asked the students to indicate their level of agreement with the statements listed in Table 3 using a 7-point Likert scale (from 1 to 7). We also asked students to provide free text comments on the best and the worst features of Mumuki.

In general all students strongly agree with the statements evaluated by the TAM model, predicting that the students would like to continue using Mumuki after the course is over. In particular both groups of students strongly agree with the statements "Mumuki improves my abilities as a Haskell programmer" (6.24 in CS1 and 6.32 in CS2) and "I think that Mumuki is easy to use" (6.44 in CS1 and 6.61 in CS2).

Regarding usefulness, the value that is lower than the others is CS2 students' average value for "Mumuki offers help when I find a task difficult", at 4.53. The students explained that "When I had a syntax error, sometimes I did not understand the explanation given

Statements evaluated (translation from Spanish)	TAM reference	CS1	CS2
1. Mumuki improves my abilities as a Haskell programmer	Improvement	6.24	6.32
2. Mumuki enables me to organise my learning time	Productivity	5.76	5.55
3. Mumuki helps me solve tasks more quickly	Efficiency	5.56	5.74
4. Mumuki offers help when I find a task difficult	Helpfulness	5.52	4.53
5. Mumuki helps me solve tasks correctly	Effectiveness	5.36	5.21
6. Mumuki is useful for learning to program in Haskell	Usefulness	6.16	5.82
7. Learning to use Mumuki was easy for me	Familiarity	6.20	6.74
8. It is easy to get Mumuki do what I want it to	Manageability	4.56	5.24
9. The interaction with Mumuki is clear and understandable	Clarity	5.40	6.08
10. The interaction with Mumuki is flexible	Flexibility	6.28	6.45
11. It was easy for me to become skillful at using Mumuki	Proficiency	6.00	6.37
12. I think that Mumuki is easy to use	Ease of use	6.44	6.61

Table 3: Results of applying the Technology Acceptance Model (TAM) indicators to the CS1 and CS2 courses. The first six statements evaluate usefulness and the other are related to ease of use.

by Mumuki". This is due to the fact that Mumuki report syntax errors by showing the output of the compiler.

Regarding ease of use, the value that is lower than the others is CS2 students' average value for "It was easy to get Mumuki to do what I want it to", at 4.56. The students explained that "It was frustrating to get the same error message again on different submissions". Mumuki does not take into account the history of the interaction when producing feedback. Taking it into account and simplifying the compiler output are topics for future research.

We asked explicitly whether they planned to use Mumuki in order to learn other languages; 95% of them answered affirmatively. Among the many advantages they mention when using Mumuki, the most frequent are the following in their own words: "It tells you when your solution is wrong and you can try many times until you get it right", "The best part is that you get immediate feedback", "You can use it at any time and from anywhere at your own rhythm", "It is particularly useful when you have to miss some lectures".

6 CONCLUSIONS

In this paper we presented a web-based coding tool that is free, open-source and provides formative feedback and assessment. We find that the dropout rates are significantly lower in those courses that use the tool at two different universities.

We applied a standard model, the technology acceptance model (TAM) to analyse students' perceptions which predict, through the strong agreement from the students, that they will continue to use the technology. One of the features better valued by the students is the automated feedback which helps the students correct many errors on their own.

Our results suggest that student dropout rates may diminish significantly when using educational coding tools. Differently from traditional instruction, they allow students to study at their own pace, increasing their chances to pass at re-tests. This paper's goal is to take a further step towards formally showing the effectiveness of such tools in order to motivate their adoption.

ACKNOWLEDGMENTS

This work was partially funded by grants PICT-2014-1833, PDTS-CIN-CONICET-2015-172.

REFERENCES

[1] Valerie Barr and Deborah Trytten. 2016. Using Turing's craft CodeLab to support CS1 students as they learn to program. *Association for Computing Machinery Inroads* 7, 2 (2016), 67–75.

[2] Peter Brusilovsky, Stephen Edwards, Amruth Kumar, Lauri Malmi, Luciana Benotti, Duane Buck, Petri Ihantola, Rikki Prince, Teemu Sirkiä, Sergey Sosnovsky, et al. 2014. Increasing Adoption of Smart Learning Content for Computer Science Education. In *Proceedings of the Conference on Innovation & Technology in Computer Science Education (ITiCSE-WGR)*. ACM, 31–57.

[3] M. Y. Chuttur. 2009. Overview of the Technology Acceptance Model: Origins, Developments and Future Directions. *Sprouts: Working Papers on Information Systems* 9, 37 (2009).

[4] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the International Conference on Functional Programming (ICFP)*. 268–279.

[5] Alex Daniel Edgcomb, Frank Vahid, Roman Lysecky, Andre Knoesen, Rajeevan Amirtharajah, and Mary Lou Dorf. 2015. *Student performance improvement using interactive textbooks: A three-university cross-semester analysis.* American Society for Engineering Education.

[6] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. 2016. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education* (2016), 1–36.

[7] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. 2012. An interactive functional programming tutor. In *Proceedings of the Conference on Innovation & Technology in Computer Science Education (ITiCSE)*. 250–255.

[8] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, et al. 2015. Educational Data Mining and Learning Analytics in Programming. In *Proceedings of Innovation & Technology in Computer Science Education Conference (ITiCSE-WGR)*. ACM, 41–63.

[9] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the Conference on Innovation & Technology in Computer Science Education (ITiCSE)*. ACM, 41–46.

[10] Amruth N. Kumar. 2008. The Effect of Using Problem-solving Software Tutors on the Self-confidence of Female Students. *Special Interest Group in Computer Science Education (SIGCSE) Bulletin* 40, 1 (March 2008), 523–527.

[11] Amruth N. Kumar. 2015. The Effectiveness of Visualization for Learning Expression Evaluation. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 362–367.

[12] Richard Lobb and Jenny Harlow. 2016. Coderunner: A Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (Feb. 2016), 47–51.

[13] Nick Parlante. 2017. CodingBat. http://codingbat.com. (2017). [Online; accessed 17-August-2017].

[14] Jaime Spacco, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. 2015. Analyzing Student Work Patterns Using Programming Exercise Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, New York, NY, USA, 18–23.

[15] Christopher Watson and Frederick W.B. Li. 2014. Failure Rates in Introductory Programming Revisited. In *Proceedings of the Conference on Innovation & Technology in Computer Science Education (ITiCSE)*. 39–44.