Automatic assessment of elementary Standard ML programs using Ceilidh

Sandra Foubister[†] Greg Michaelson[‡] Nils Tomes[†]

[†]Learning Technology Centre
[‡]Department of Computing and Electrical Engineering
Heriot-Watt University
Riccarton, Edinburgh EH14 4AS

S.P.Foubister, G.J.Michaelson, N.Tomes@hw.ac.uk

[‡]Primary contact.
Submitted to Journal of Computer Assisted Learning.

Abstract

Ceilidh is an interactive environment which supports computer programming course organisation, practical work and assessment. This paper describes its use to support a first level programming course for the functional language Standard ML. Automated program assessment systems are surveyed and the general Ceilidh approach to automatic assessment is discussed. Extensions to Ceilidh to assess Standard ML programs are considered and an evaluation is made of the effects of its use on student learning. The main conclusion is that Ceilidh use significantly reduces the burden of marking on the lecturer, while not affecting the overall level of achievement of the students. Keywords:

automatic assessment; functional programming; Standard ML; Ceilidh

1 Introduction

Assessment is fundamental to education. Formative assessment is crucial to a student's learning in that it provides feedback on progress, allowing them to pace their studying and to focus on areas of difficulty. Timely feedback may help to detect misconceptions, which is particularly important to the weaker or less-confident student. Summative assessment is generally used as a measure of the extent to which learning criteria have been achieved.

Traditionally, assessment falls upon educators. However, a significant proportion of a teacher's time can be taken up with repetitive marking, as is increasingly the case with expanding class sizes, with no concomitant increase in resources. This consumes staff time which might be better redirected to other, higher-value, aspects of teaching students. The increased burden on educators can only be reduced if either assessment itself is reduced, or innovative methods are introduced. The solution discussed here is to automate assessment.

There are also well known difficulties in ensuring consistency and accuracy when hand-marking assignments, particularly if these are submitted in printed form. Lack of repeatability in marking can allow unintended biases to creep into how different groups of students are assessed. Thus the introduction of automated assessment may not only reduce the burden of marking, but also addresses the problem of such "unfairness".

We have been investigating the automatic assessment of computer programs as part of the UK Higher Education Funding Councils' Teaching and Learning Technology Programme (TLTP) Ceilidh consortium. The original Ceilidh system was developed at the University of Nottingham to support assessment and organisation of imperative programming courses in C. We have extended Ceilidh to provide support for functional programming in Standard ML, and have evaluated its impact on exam performance.

2 Automatic program assessment systems

For effective assessment, it is vital to enunciate clear criteria and to develop appropriate instruments to test them. In general, such criteria may be *absolute*, typically involving the evaluation of something in terms of ideal features, or *relative*, typically involving comparison between something and a model. For automatic assessment by computer, the assessed material must be in machine manipulable form and the assessment criteria must be expressible in terms of symbolic manipulation.

In the assessment of programming, criteria seem straightforward to formulate:

- Is the program correct, *i.e.* does it do what it is supposed to do?
- Is a program optimal *i.e.* how well does it do what it is supposed to do?

Programming is well suited to automatic assessment. Programs are necessarily in symbolic form for computer manipulation, and the above criteria have straightforward expression as symbolic processes. These criteria, and systems to assess them, are now considered briefly in turn.

2.1 Automatic correctness assessment

Approaches to assessing program correctness are invariably based on *relative* criteria. Formally, a program is correct if it can be proved to meet its specification. Alas, program proof is recursively undecidable and heuristics based theorem provers are not yet mature enough for general purpose use. Thus, most correctness assessment is empirical and based on checking a program's outputs from inputs.

Traditionally, students are either supplied with inputs or make up their own. In either case, they return the outputs to educators for inspection by eye. In computer based systems, student program outputs are compared automatically with model outputs. Typically, students submit a program which the system runs to produce the output. Inputs may be fixed or randomised. The model output may either be fixed or generated from a model program.

Automatic program assessments systems have a long pedigree. One of the earliest was developed in the late 1950's by Hollingsworth [Hol60] for IBM 650 machine code programs. This ran batches of student programs from cards embedded inline within a grader, an assessment program, to provide initial variable values and make absolute checks on correct final values. Hollingworth believed that the use of the grader was more effective for student learning than laboratory based programming teaching, saved time and money, and enabled teaching of large numbers of students. Such considerations are still a major motivation for the use of automatic assessment.

Many early applications were for the assessment of numerical analysis programs. Naur [Nau64] describes an early 1960's experiment in grading GIER ALGOL programs from paper tape, again embedded inline within a grader to perform problem specific behaviour and absolute output analysis. Forsythe and Wirth [FW65] also describe two ALGOL graders from 1965. In both cases, the student program was embedded as a procedure in a problem specific grader program. The first, for Burroughs 220 ALGOL, generated random test data and compared student program output with that from a specimen answer. The second, for IBM 7090 ALGOL, is similar to Naur's. Subsequently, Berry [Ber66] replicated both Naur's, and Forsythe's and Wirth's studies and criticised their poor feedback to students. He then applied their approach to a wide range of numerical analysis problems, again in ALGOL.

Braden's and Perlis' 1965 TEACH system [BP65] was developed for ALGOL-20. Here the student's program calls a TEACH procedure to compare the results of a model answer with the student's. Temperly's and Smith's 1968 system [TS68] for PL/1 also calls the grader from within the student's program, using a model answer to generate comparison results. Hext's and Winings' 1969 Basser Automatic Grading System (BAGS) on the KDF9 [HW69] reflected the growing maturity of operating systems. BAGS was an operating system utility and ran exercises as normal batch jobs. Programs could be run on a fixed set of test data, either with a nominated or a random test. Outputs were compared with preset constants. BAGS was used with ALGOL, an ALGOL subset called MINIGOL and an assembly language I/O package.

In the 1970's and early 1980's the focus moved away from correctness to optimality assessment [Mic96]. Since then, there has been renewed interest in automatic correctness assessment. However, all the elements of contemporary systems may be found in those from the 1960's.

Carrington's, Robinson's and Whale's 1984 `give` system [CRW84] runs under UNIX. Student programs are tested with standard data and results are evaluated through a lecturer defined process. This could compare them with known outputs or with specimen program outputs. Benson's 1985 system [Ben85] uses problem specific test drivers to assess student programs. Note that Whale [Wha91], citing Benson, is sceptical of attempts to automate all aspects of assessment as "only simple performance factors are able to be assessed". Isaacson's and Scott's 1989 system [IS89] also runs under UNIX and compares student's answer output with known outputs. They suggest that specimen programs could be used to generate comparison outputs within their system. Most recently, Pardoe's and Vicker's 1994 SPROUT system [PV94] for Turbo Pascal under MSDOS compares student program outputs with known outputs. Webber's 1996 Pascal Trainer [Web96] is based on problem specific tests written in Pascal which linked automatically into a student program. Hurst's 1996 system [Hur96] under UNIX is based on literate programming. It assesses an answer program through problem specific test scripts using random or generated test data.

2.2 Automatic optimality assessment

In early automatic systems, optimality assessment usually focused on program speed and accuracy of results. Now, there is more concern with assessing program style in correct programs or on identifying the source of mistakes in incorrect programs.

Michaelson [Mic96] reviews style criteria and automatic style assessment systems, using a linguistic taxonomy of program style. In these terms, most style assessment criteria are *absolute* and based on lexical (*e.g.* meaningful identifiers), syntactic (*e.g.* layout) or pragmatic (*e.g.* comments, assertions) considerations. Automatic semantic style assessment of functional programs is discussed briefly below.

In contrast, many approaches to mistake identification involve *relative* criteria. Lutz [Lut93] reviews automatic program understanding and debugging systems, many of which are based on extracting a plan inherent in a program and comparing it with a model plan for solving a problem. Bental [Ben93] also reviews systems which may be used to critique students' programs, and discusses techniques for analysing data structure as well as algorithm design choices. Mistake identification is not considered further here.

3 An overview of Ceilidh

Ceilidh [BBF+93] was originally developed at the University of Nottingham, to assist the teaching of the imperative languages C and C++, under the Enterprise in Higher Education Initiative. In principle, Ceilidh's general framework could support any form of symbolic assessment. However, Ceilidh has been used almost exclusively for programming assessment and has many features found in the systems discussed above. Specifically, it is best oriented to relative correctness marking, where student answers are tested on a fixed repertoire of inputs, and outputs are compared with those from a model solution. In contrast to other systems, Ceilidh is not language specific.

Ceilidh is an interactive system which also provides a wide degree of support for course organisation. In particular, tutors may monitor the progress of both individuals and a whole class in carrying out practical exercises.

Within Ceilidh, courses are structured into units with associated series of exercises. For each exercise there is a *question*, describing the programming constructs to be defined, together with examples of appropriate outputs for illustrative inputs. A *skeleton answer* helps guide the student to the correct answer. This emphasises important general elements in the early exercises, and special features needed for later ones. A *model answer* is provided both for the student to use, without seeing its text, against standard test input, and later for reference. The model answer is also used in the marking process. A set of *test data* allows the student to compare the result of applying their answer to particular tests with that of the model answer. A *marking scheme* includes both dynamic marking and appropriate analysis of the answer text. *Feedback* helps the student identify any areas where they have lost marks.

Each Ceilidh exercise has its own marking scheme. This may exploit features inherent in the system, such as the *oracle* program which uses pattern matching of text, but may also incorporate marking methods defined by the tutor. Each element of the mark is weighted, and also has associated with it a short description that is used in feedback to the students.

When students are satisfied with their answer to an exercise, they can submit it to the system. Resubmissions are possible, though there is a facility for the tutor to limit the number of submissions.

The student gets immediate feedback, being allocated a mark at each submission. The mark is broken down into the various categories predetermined by the marking system for that exercise. This focuses the student on any area of deficiency.

Each time a student submits an exercise, the overall mark allocated is recorded. These marks, together with the information of whether or not students have indeed submitted particular exercises, allow the tutor to identify potential problems early on, and to monitor the progress of individual students.

In addition to normal email channels, students can email tutors from within Ceilidh in the context of a particular exercise. They have the option of including the code that they have submitted, for reference.

The lecturer can set groups of exercises at appropriate points in the course by making them available to the students through Ceilidh. They can then be used by the students until such time as they should more usefully be working on the next group, at which time the individual unit may be *closed i.e.* the students may no longer submit answers.

Once an exercise has been closed, the lecturer may make the model program available for inspection, rather than having to distribute model answers. These may then be used by the students to reinforce their understanding of particular points, and used in their preparation for the written exam.

All course notes and exercise text are available on line. It is thus possible for a course based on Ceilidh to be "paperless". In practice it is more convenient to distribute course notes and exercise sheets as in the traditional style. This ensures that all students (who attend lectures) have a copy, and lets them study away from the machines. They can then, for example, refer to the on-line versions when they have not brought the paper versions to lab sessions, and print them out as necessary.

4 Ceilidh for Standard ML

In the early 1990's, Ceilidh was made available for use by other universities and further developed to support a wider variety of programming languages with funding from the UK Higher Education Funding Councils' Teaching and Learning Technology Programme. Our group was responsible for the introduction of a version for functional programming in Standard ML (SML) [MTH90].

Ceilidh for SML is based on a traditional programming course, taught to first year students who have already had one term's experience of an imperative language. Some of the more advanced features of SML are not covered, because of the introductory nature of the course. There is thus scope for a subsequent second level course.

Within Ceilidh for SML, the course of lectures, the lecture notes and the exercises are unchanged in content but the course was reorganised into five units. Each unit has associated with it around 10 exercises, each requiring the student to define an SML function, or group of functions, to illustrate various concepts introduced in the course. A student who tackled all the exercises would write 90 different SML functions. In the traditional version of the course the exercises were distributed on paper, and handed in to the lecturer for hand marking. In the new version the lecturer merely refers the students to the appropriate unit of exercises in Ceilidh.

Our use of Ceilidh for SML is to support self-paced practical work. While some laboratory sessions are timetabled with a demonstrator available, students may otherwise access the materials when they wish. In order to motivate students to use Ceilidh, the marks obtained are included in the final assessment. Unless resubmission is controlled, it is theoretically possible for everyone to get full marks for this element of the course. Because of this, and because it is virtually impossible to detect plagiarism in the elementary exercises used, there is still a complementary written exam at the end of the course, counting for 70% of the overall mark.

4.1 Marking and feedback

The Ceilidh for SML system assesses functions for both correctness and optimality. Correctness assessment is as described above. Note that there will usually be more than one test for each exercise. A typical SML function is recursive and processes a sequence of values. Such sequences may be empty (base case) or non-empty (recursion case): the behaviour of a function should reflect this. Thus, tests will be present to exercise base and recursion cases, and also to check for correct processing of particular values in sequences. Within the overall percentage allocated for such dynamic marking, each sub-test may be differentially weighted.

Optimality assessment has components for style and type. The style checker applies absolute tests to a function's structure to ensure the appropriate use of constructs. In particular, it checks that a recursive function has explicit base and recursion cases to relate directly its form to that of the data sequence it processes. Such checks are semantics oriented: the original Ceilidh performed relatively crude lexical, syntactic and pragmatic tests based on textual feature counting.

A SML function may be characterised as a mapping from its inputs to its outputs, in terms of the inputs and outputs types. For example, a function called `bananacount` might return an integer count of how often the string `"banana"` appears in a list of strings *e.g.*

```
bananacount ["apple","banana","cherry"] ==> 1
```

Thus, it would have type:

```
string list -> int
```

An SML implementation can deduce automatically a function's type. However, we think that a student's ability to provide that type explicitly is an indication of their understanding of function's purpose. Thus, all the SML Ceilidh exercises ask for a type specification to be given for each function as a comment. The text of the comment is then compared with the equivalent specification for the model answer, and a proportion of the mark is allocated for a correct correspondence. Early versions of Ceilidh for C count lines of comment, without looking at the content, to contribute the overall mark. Here, the nature of the content is paramount.

Finally the marking of an exercise may use the built in oracle program to look for the presence or absence of specific features in a function definition. For example, the student may be required to define one function in terms of another or to define a function without using a predefined function.

5 The evaluation of Ceilidh for SML

The resulting system was used to manage the practical work in the department's existing SML course, and the impact of this change on student performance was assessed over four years. This was facilitated by the availability of exam marks for the final year of the traditional version of the course. These are directly comparable to the final exam mark component of the Ceilidh enhanced course, so constitute control data. The students' intake qualifications were also recorded, to allow variations in intake profiles from year to year to be taken into account.

For clarity, this evaluation discards the data from the first experimental year, when the Ceilidh system was first introduced, using the data from the subsequent two years when the use of the system had stabilised. The evaluation reported here relates to 158 students, 50 of whom were in the control group.

Source of variation	*Sum of squares*	*df*	*Mean square*	*F*	*Significance of F*
Main effects	8149	4	2037	3.708	0.007
entry qualification level	7675	3	2558	4.657	0.004
experimental/control group	614	1	614	1.119	0.292
Interaction of entry qualification level and experimental/control group	1503	3	501	0.912	0.437
Explained	11846	7	1692	3.080	0.005
Residual	82413	150	549		
Total	94260	157	600		

Table 1: Analysis of variance in the final examination mark obtained, by experimental teaching group and entry qualification level.

As shown in Table 1, student achievement by the control and by the experimental groups was statistically similar, with the final exam score being related to the students' intake ability rather than to the style of formative assessment which they received. Further analysis shows that students across the ability spectrum appeared to be equally well served by Ceilidh, with no significant relationship between the level of entry qualification and performance under the different assessment conditions.

6 Conclusions

The use of Ceilidh for SML appears to offer a number of benefits. summarised.

From the lecturer's point of view, marking time for each student exercise script was saved, multiplying up to a significant component of lecturer time over each term. This saving in marking time is estimated at around one and a half hours per student, over the whole course, based on an average of 1 minute to mark each of the 90 functions by hand. For a typical class of 80 students, this represents a saving of around 120 hours over a 10 week course. It is encouraging that this saving is not at the expense of student achievement.

Furthermore, a lecturer may monitor whether or not a student has attempted an exercise, how many attempts a student has made at each exercise and the rate at which they are progress through the practical work. This enables early identification of students who may be at risk of failing and in need of additional support.

Students like the swift feedback which they receive on their performance, and this helps to build their confidence in their developing programming skills.

The email facility within Ceilidh allows students to contact advisers with exercise specific problems, and to include their code in their query. This enables relatively quick lecturer feedback to complement that offered by the system itself. Bental [Ben95] provides a detailed analysis of student email use within Ceilidh for SML.

Finally, students perceive the Ceilidh system as fair, complementing the intention to standardise marking and remove the scope for unfair variations, for example as might occur in an individual lecturer's marks.

There are also a number of weaknesses in Ceilidh. The interface is rather old fashioned, based on command line interpretation: a windows based version is under development. Furthermore, Ceilidh does not provide a complete programming environment. Thus students must simultaneously become familiar with Ceilidh, the underlying operating system, the SML language and and the SML support system. This tends to overload them early on in the course.

Many students take advantage of the facility to resubmit exercises to improve their mark, after taking note of the feedback given. A small minority make apparently blind changes to exercises without adopting an appropriate program analysis or testing strategy. Facilities exist within Ceilidh for a lecturer to limit the number of attempts at any exercise, to control the time students spent on such unproductive work. It has not been felt necessary to make use of these: the vast majority of students adapt a sensible approach to program development and accept a good pass mark without searching for a few elusive additional percentage points.

Following the preliminary evaluation of student performance, the Ceilidh system has now become a regular component of Heriot-Watt University's introductory SML course. Ceilidh courseware can be obtained from the WWW at http://www.crg.cs.ac.uk/ceilidh/ .

Acknowledgements

We would like express our gratitude to the UK Higher Education Funding Councils' Teaching and Learning Technology Programme for funding the work described here. We would also like to thank Steve Benford, Eric Foxley and Neil Gutteridge from the University of Nottingham for constructing the original Ceilidh system and for their support during the development of Ceilidh for SML. Finally, we wish to thank the first year Heriot-Watt University students who participated in the study as part of their degree work.

References

[BBF+93] S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A. M. Zin. Experiences with the Ceilidh system. In *Proceedings of the International Conference in Computer Based Learning in Science*, Vienna, 1993.

[Ben85] M. Benson. Machine assisted marking of programming assignments. *ACM SIGCSE Bulletin*, 17(3):24–25, September 1985.

[Ben93] D. Bental. *Recognising the design decisions in Prolog programs as a prelude to critiquing*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1993.

[Ben95] D. Bental. Why doesn't my program work? requirements for automated analysis of novices' computer programs. In *Workshop on Automated Understanding of (Novice) Programs, World Conference on AI and Education, Washington DC, USA*, 1995.

[Ber66] R. E. Berry. Grader programs. *Computer Journal*, 9(3):252–256, November 1966.

[BP65] R. T. Braden and A. J. Perlis. An introductory course in computer programming. Technical report, Carnegie Institute of Technology, June 1965.

[CRW84] D. Carrington, K. Robinson, and G. Whale. Give: a system for collecting and testing student assignments. In *Proceedings of 7th Australian Computer Science Conference*, pages 21–1 to 21–10, February 1984.

[FW65] G. E. Forsythe and N. Wirth. Automatic grading programs. *Communications of the ACM*, 8(5):275–278, May 1965.

[Hol60] J. Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3:528–529, October 1960.

[Hur96] A. J. Hurst. Literate programming as an aid to student marking. In J. Rosenberg, editor, *Proceedings of 1st Australasian Conference on Computer Science Education*, pages 280–286. ACM Press, July 1996.

[HW69] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Communications of the ACM*, 12(5):272–275, May 1969.

[IS89] P. C. Isaacson and T. A. Scott. Automating the execution of student programs. *ACM SIGCSE Bulletin*, 21(2):15–22, June 1989.

[Lut93] R. Lutz. *Towards an intelligent debugging system for Pascal programs: on the theory and algorithms of plan recognition in Rich's plan calculus*. PhD thesis, Cognitive Science Research Papers 272, University of Sussex, 1993.

[Mic96] G. Michaelson. Automatic assessment of functional program style. *Proceedings of 1996 Australian Software Engineering Conference*, pages 38–46, 1996.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge MA, 1990.

[Nau64] P. Naur. Automatic grading of students' programs. *BIT*, 4:177–188, 1964.

[PV94] J. Pardoe and P. Vickers. Using a prototype program assessment tool. In *2nd All-Ireland Conference of Teaching of Computing*, pages 232–236. Computing CTI, September 1994.

[TS68] J. F. Temperly and B. W. Smith. A grading procedure for PL/1 student exercises. *Computer Journal*, 10(4):368–373, February 1968.

[Web96] A. B. Webber. The Pascal Trainer. In K. J. Klee, editor, *Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education*, pages 261–265. ACM SIGCSE, February 1996.

[Wha91] G. Whale. Paperless assignment marking. In *Proceedings of 14th Australian Computer Science Conference*, pages 32–1 to 32–6, February 1991.