Fully Automatic Assessment of Programming Exercises

Riku Saikkonen, Lauri Malmi, and Ari Korhonen
Department of Computer Science and Engineering
Helsinki University of Technology, Finland
{rjs,lma,archie}@cs.hut.fi

Abstract

Automatic assessment of programming exercises has become an important method for grading students' exercises and giving feedback for them in mass courses. We describe a system called Scheme-robo, which has been designed for assessing programming exercises written in the functional programming language Scheme. The system assesses individual procedures instead of complete programs. In addition to checking the correctness of students' solutions the system provides many different tools for analysing other things in the program like its structure and running time, and possible plagiarism. The system has been in production use on our introductory programming course with some 300 students for two years with good results.

1 Introduction

Introductory programming courses often have lots of small exercises, designed to teach students to write small programs first. After solving them the students can move on to solve larger assignments. Such course organisation has also been used in Helsinki University of Technology on the basic programming courses.

The course given for students studying computer science as their major is based on using the functional programming language Scheme [1,6]. Scheme is not widely used in "practical programming", but there are over 250 colleges, universities, and secondary schools that are using Scheme in their curricula [7]. We use it on our course because Scheme provides better possibilities to present and teach many important and complex concepts in computer science than more commonly used programming languages like Java or C. Unfortunately, many practically oriented students have not been very motivated to learn an academic language. Therefore we have used small but mandatory weekly programming exercises in order to keep the students active during the whole course instead of trying to catch on the course just before the exams.

In 1997, due to the rapidly increasing number of students, we did not have enough resources to assess the exercises manually. We had to stop assessing them and make the exercises voluntary.

ITiCSE 2001 6/01 Canterbury, UK

This was a serious drawback for the results of the course, because voluntary exercises did not very well support the prementioned aim of keeping students busy.

In order to return to the previous convention, the grading load needed to be cut down heavily. One solution to this problem is to assess the exercises automatically or semi-automatically. Many such systems have recently been presented for programming courses [2,4], data structures and algorithms courses [5] and others [3]. We have used the Ceilidh system [2] to assess exercises written in C or Java on the basic programming course for students studying computer science as their minor since the year 1994.

The main principle of Ceilidh is to use string matching to compare the output of the student's program with the model output given by the teacher. This approach is not very suitable for our Scheme course, because in most exercises students write single Scheme procedures instead of complete programs. This is closer to the idea of implementing algorithms than to writing programs. Moreover, it seems unimportant to require the students to write trivial code just for reading input and writing output values. Therefore we would have had to make artificial modifications to the exercises, which we did not want to do.

One solution would have been to use a semi-automatic approach such as presented by Jackson [4]. This approach requires the teacher to monitor the assessment process and to do some of it manually. However, even this would cause too much work on large courses with lots of exercises.

To solve the problem we developed *Scheme-robo*, an automatic assessment system for Scheme exercises. It assesses exercises completely automatically without human intervention.

Scheme-robo has a number of features that we consider important. First, the assessment is carried out on-line, so that students get their results almost immediately and can resubmit a failed exercise after reconsidering their solution. Limiting the number of submissions forces them to think about the reason for the failure instead of plain trial-and-error debugging. Second, we avoid the problem of comparing the expected result with slightly different output from student code [4], since most of the exercises are Scheme procedures that return a value instead of printing it out. Return values can be analysed within the Scheme language, usually by simply comparing them to the correct answer using a standard equality predicate. Third, the Lisp-like syntax enables us to easily analyse the structure of the students' code. This is very useful in assignments where students fill in the given exercise code. The same feature can be used to detect plagiarism, as well. Fourth, we can set restrictions on what language constructs they are allowed to use for particular exercises. Fifth, Scheme-robo allows us to use randomised tests. Finally, the use of Scheme allows us to run code in a fully safe "sandbox", so that malicious or non-terminating code does not harm the assessment. We can actually use the run time control even for rough run time analysis,

i.e., if a linear time solution is required, our system catches possible $O(N^2)$ algorithms.

The system has been in production use for two years, with about 60 exercises and 300 students per year, and it has worked suprisingly well. All exercises require the student to write either Scheme code or short verbal answers (for example, complexity figures).

The structure of this paper is the following. In the next section we explain briefly how students use the system. In Section 3 we consider more closely how the student code is analysed in various ways and give an example of setting up an exercise. In Section 4 we present a general view of the system architecture and in Section 5 we discuss some results and observations we have made during these two years. The final section includes some final remarks and ideas for future work.

2 Student's point of view

Students submit their solutions to Scheme-robo via e-mail. The system automatically responds to them with a receipt containing a copy of their submission and a set of points and comments to the submitted exercises.

Currently Scheme-robo responds within about ten minutes. While small response times are usually preferred, in this case an instant response would encourage the students effectively to debug their code in the Scheme-robo system by trial and error. Ten minutes was selected to encourage the students to test their code by themselves before submitting it.

We have a fixed deadline for finishing each set of exercises, and the students may submit a solution for every exercise for up to 20 times. In practice, most students get it right with the first few tries.

New exercises are automatically sent to the students via e-mail. The students can also request copies of the exercises and the report of their current points by sending e-mail to the Scheme-robo system.

3 Teacher's point of view

To add a new exercise into the Scheme-robo system, the teacher needs to write a *configuration file* that describes how the solutions are assessed. The content of the file depends on the type of assessment. Usually the files are about 20 to 100 lines long and contain test runs, model solutions and/or other information.

Scheme-robo assesses solutions in several ways. The main method is to execute the code using test runs specified by the teacher and to examine the return values. The system can also do some analysis on the structure of the submitted code, e.g., to make sure that it conforms to a skeleton given in the problem statement. In addition, the system can check for specific keywords in the code. Finally, simple pattern matching based on regular expressions is used in such exercises where students do not write code. For example, the problem might ask for a set of numbers or a complexity figure (e.g. $O(N)$), or it could be simply a multiple-choice question.

Each of these assessment methods has its own type of configuration which is described below more in detail. Moreover, another configuration file can be included into the current one. This feature is used for combining different methods, for example, when analysing code structure before executing test runs. It is also used for sharing parts of the configuration between different exercises, e.g., a set of test runs for all exercises having to do with a particular topic.

In order to help in testing the configuration files, we have written a set of simple scripts that automatically check whether our model answers are passed by the configuration files.

The Scheme-robo system gives points for each executed test. Points from individual tests are summed together to get the total points. There is also a possibility to give a `fail` value for the number of points, which means that the exercise is immediately rejected, without executing further tests. In practice, we have assessed the exercises simply as *passed* or *failed.*

3.1 Test runs

The main method of assessing student code is to execute test runs and examine their results. All test runs are independent of each other.

Most of the test runs are "fixed"; that is, expressions are given by the teacher and tested with every student. For example, we can test if the Scheme expression `(fact 5)` returns 120, which is the factorial of 5, calculated using a `fact` procedure submitted by the student.

A general problem in automatic assessment is the difference in wording and layout in students' answers. For example, one program could print out "The factorial is 120." while another could say "5! = 120". We are able to avoid this problem by looking at return values from procedures instead of the actual output strings. Moreover, the students are also spared from the pedagogically unimportant effort of writing output routines.

In addition to fixed tests, there is a possibility to include randomness in the tests. Consider, for example, an exercise where students implement a sorting algorithm. We can specify a test run that generates five random lists of numbers, sorts them using the student's code, and compares the result to what the model solution returns for the same inputs.[1]

Many exercises require the use of code given in the textbook. We therefore have the facility of including a fixed set of definitions to be executed before the student's code when doing the test runs. This feature is useful for various supporting procedures, or when students need to modify large sections of code and it is more convenient to submit only the modified procedure definitions.

3.2 Testing program structure

Keyword search For some exercises, we want to disallow the use of certain Scheme primitives. For example, there is an exercise to reimplement the `reverse` primitive that reverses a list. A solution that uses the same primitive to implement itself should naturally not be accepted.

The list of keywords to be rejected is set up in the configuration file. This is practical, because Scheme does not have a large set of primitives. For instance, we can require a purely functional implementation of an exercise simply by disallowing primitives `set!`, `set-car!` and `set-cdr!`.

Analysis of program structure In addition to keyword search, Scheme-robo can also do some simple analysis on the structure of student code. Scheme or Lisp code is trivially reduced into a list structure which represents a kind of abstract syntax tree. This tree can be examined to look for particular subpatterns or some specific general structure.

In practice, this feature has mostly been used to check whether the student has followed a mandatory skeleton given in the

[1] A model solution is actually needed only if there are random tests.

problem statement.

Detecting plagiarism Plagiarism detection is important when assessing exercises automatically. Some preprocessing of submitted code is carried out before this. Individual solutions are reduced to a kind of abstract syntax tree. Variable names are removed and, for example, definitions and arguments of commutative primitives are sorted in a particular order. The resulting trees are finally compared to each other.

We observed that our exercises are too small for detecting plagiarism from individual exercises. Therefore the system compares all exercises, looking for pairs of students that have many similar solutions. A plagiarism estimate is given for each pair of students, and suspicious pairs are examined manually.

3.3 Feedback to the student

The student is given various kinds of feedback for his solution. Each test has a set of possible outcomes given in the configuration file, for example, a set of possible answers for a test run. The configuration file specifies points and comments for each outcome.

The comments are thus given on the basis of individual tests. In practice, we have mostly used them as error messages: the most common comments identify the test run that failed. Comments could also be used as warnings ("this didn't work, but that's ok") or as encouragement (when the student's solution includes something more than what was required in the problem statement).

3.4 An example

One of the first exercises on our course has been to write a procedure to compute the cube root of its argument using a method given in the textbook (exercise 1.8 of [1]).

The exercise requires students to use internal definitions (i.e., block structure). This is checked via structural analysis. In the following configuration file, `??` represents any symbol or list structure, and `??*` zero or more such. The solution is rejected if internal definitions are not found. Otherwise one point is given.

```
(if (structure
     (??*
      (define (cube-root ??)
        ??*
        (define (?? ??*) ??*)
          ??*)
      ??*))
    (1)
    (fail "No internal definitions."))
```

The Scheme primitive `expt` must not, of course, be used in the solution. This is tested by the following code, which either rejects the solution or gives 0 points.

```
(if (keyword expt)
    (fail)
    (0))
```

Test runs can be specified as follows. Calculating the cube root of 125 should return 5.0; the configuration below gives one point for this, and otherwise rejects the solution with the comment identifying the procedure call that did not work.

```
(test (cube-root 125)
      (r 5.0 1)
      (else fail "*expr* doesn't work"))
```

This is the essence of what is necessary for a configuration file. The configuration actually used for the cube-root problem contains some more test runs (including random ones); it has 29 lines plus empty lines and comments. Creating and testing the configuration for a new exercise usually takes less than an hour for these kinds of small exercises.

4 Architecture

Scheme-robo reuses the core of the TRAKLA system [5] to handle orthogonal tasks of course administration: handling the submitted answers, keeping track of points, calculating statistics, etc. The assessment itself is implemented as a "checker module" for TRAKLA. The module receives as input a submitted answer to a single exercise and returns a number of points as its result. Short configuration files are used to specify test runs and other information specific to each individual exercise.

4.1 Executing student code

Student code is executed in a safe "sandbox", a metacircular Scheme interpreter specifically created for this purpose. The interpreter was based on the analysing metacircular Scheme interpreter from [1], and it has been extended to implement most of the Scheme language specification [6].

The interpreter has some special features that distinguish it from a run-of-the-mill Scheme interpreter. In order to avoid infinite loops in student code it includes a count of the number of evaluations done when running a particular test; when this count reaches an exercise-specific maximum, the submitted solution fails. The same count can also be used for very coarse complexity analysis. For example, we can measure whether an algorithm runs in approximately linear time by specifying a limit for the number of evaluations that is difficult to achieve with, for example, an $O(N^2)$ algorithm.

The customised metacircular interpreter provides a safe sandbox, because we did not include Scheme features for file I/O or other things that might compromise security when untrusted student code is executed. These features have not been necessary in practice in our exercises.

5 Observations from using the system

The automatic assessment has worked quite well and we have been able to assess all small exercises automatically.[2] In practice, the system has been stable and has required very little manual administration.

Most of the problems that students have reported with Scheme-robo are due to additional non-standard features of the Scheme implementation we use on the course. The problems have been relatively minor, and we have been able to solve them by adding support for such features.

In addition to assessing the exercises, Scheme-robo keeps extensive logs of its operation. This data is useful for monitoring the progress of the students and, for example, the difficulty of individual exercises. On the fall 1999 course, we divided our exercises into five "rounds", so there were deadlines every two weeks. The logs clearly confirmed the assumption that many students do exercises just before deadlines. For instance, about 1000 exercises per day were submitted before the first deadline, and only about 60 per day just after it.

[2] There is also a larger programming project on the course, which is assessed manually.

5.1 The students' opinion

We had around 350 students on our course in the autumn term 2000. The exercises were divided in 5 two-week rounds, each having 12 exercises, 4 of which were mandatory. The students completed, on average, 5 out of 12 exercises. The voluntary exercises gave extra points for the exam.

In October 2000, we asked students to give feedback on the automatic assessment system. Out of 229 respondents, 80 % thought that automatic assessment in general is a good or excellent idea. The Scheme-robo system was also praised: only 10 % of the students thought that it assessed exercises badly. The rest thought that the Scheme-robo system assessed exercises moderately well (41 %), well (44 %) or excellently (6 %).

We also asked about an area that we knew needed improvement: only 45 % of the students thought that the error messages (comments) given by Scheme-robo are adequate or good.

6 Discussion

There are some things that need to be taken into consideration when creating exercises that are automatically assessed. First of all, exercises must be well-specified to the level of specifying an order for function arguments etc.

It is necessary to solve new exercises before one can be reasonably certain that the automatic assessment is done correctly. This is, of course, a good thing to do in any case, but it can often be postponed when assessing exercises manually.

The system is by no means complete. Better error messages are the most important area of improvement in it. In addition, automatic assessment of coding style would be a valuable aid. And finally, we should have a means for tailoring the exercises to be different for each student, as in TRAKLA [5]. But even now Scheme-robo has proved its value as an important tool with great pedagogical value. Without it, we could not assess the exercises and give the students enough feedback on such a large course with 300 students. Moreover, no reasonable human resources would be enough to give feedback so quickly and allow the students to resubmit their solutions and learn from their errors in this way.

We use Scheme on our course, but this approach to automatic assessment should also work well in other programming languages. Testing individual procedures instead of complete programs (thus avoiding the problems caused by I/O, as discussed above) is possible in any language with an interpreter (perhaps even in a traditional compiled language, if the assessment system writes a main() function to do the test runs). We used a custom-made metacircular interpreter, but even a normal run-of-the-mill interpreter could be used (for example, as a subprocess of the assessment system). A safe sandbox for student code is more difficult to implement in this case, but could be done by, e.g., preanalysing the code and controlling the execution time, or by using the safety features of Java. It would be interesting to see whether this approach to automatic assessment would work in Java.

7 Acknowledgements

We thank Kenneth Kronholm and Timo Lilja for writing major parts of the Scheme-robo system.

References

[1] Abelson, H., Sussman, G. J., and Sussman, J.: Structure and Interpretation of Computer Programs. 2nd Edition. ISBN 0-262-51087-1, MIT Press, 1996.

[2] Benford, S., Burke, E., Foxley, E., Gutteridge, N., Mohd Zin, A.: Ceilidh: A Course Administration and Marking System. *Proceedings of the International Conference of Computer Based Learning*, Vienna, 1993.

[3] English, J., Siviter, P.: Experience with an Automatically Assessed Course. *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pp. 168-171, Helsinki, Finland, 2000.

[4] Jackson, D.: A Semi-Automated Approach to Online Assessment. *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pp. 164-167, Helsinki, Finland, 2000.

[5] Korhonen, A., and Malmi, L.: Algorithm Simulation with Automatic Assessment. *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pp. 160-163, Helsinki, Finland, 2000.

[6] Kelsey, R., Clinger, W., and Rees, J. (eds.): Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9), October 1998.

[7] Martin, E.: Schools Using Scheme. *http://www.schemers.com/schools.html.*