A web based environment to support teaching of functional programming

Dietmar Rösner
Otto-von-Guericke-Universität Magdeburg,
(Institute of Knowledge Processing and Language Engineering),
P.O. Box 4120,
D-39016 Magdeburg, Germany
roesner@iws.cs.uni-magdeburg.de

Mario Amelung
Otto-von-Guericke-Universität Magdeburg,
(Institute of Knowledge Processing and Language Engineering),
P.O. Box 4120,
D-39016 Magdeburg, Germany
amelung@iws.cs.uni-magdeburg.de

ABSTRACT
We report on our approach to employ the WWW in order to support lecture room teaching of functional programming, especially Haskell and Scheme, with interactive web based tools for the students: immediate feedback on interactively submitted solutions of programming tasks, structured submissions of induction proofs and multiple choice questionnaires.

Keywords
Haskell, Scheme, teaching environment, automated checker

1. INTRODUCTION
In our curriculum, we have a course ‘Programming concepts and modelling’ (in German: Programmierkonzepte und Modellierung). The primary audience of this lecture (and the accompanying group exercises) are CS students in the third semester (i.e. after one year of study). For this group, the lecture is obligatory. In addition, there are students participating from other subjects in our faculty (i.e. students of business informatics, computational visualistics, and engineering informatics) as well some from the faculty of electrical engineering.

The purpose of the course is to teach the essence of different programming paradigms (functional, logical, and object-oriented) and to relate their relevant concepts to the students’ prior experience with Java or other languages (mostly imperative ones).

We are convinced that the experience and insights gained by solving programming problems in a number of exemplary languages from the different paradigms (i.e. Haskell, Scheme, and Prolog) is indispensable for the success of the students. But we observed that traditional exercise groups with paper and pencil solutions of programming tasks were not sufficiently helpful to achieve this goal. Many students tried to avoid real programming and somehow tried to hide behind the backs of their more engaged fellow students. We therefore searched for means to somehow force (or better: seduce) any single student to develop, test, and debug finally running programs in Haskell, Scheme, and Prolog. On the other hand the overhead for the assistants in checking the students’ submissions should be minimized. This motivated our attempts to automate checking and preselection of students’ submissions for mandatory programming tasks as far as possible.

In addition, we wanted to offer web based self tests in the style of multiple choice questionnaires. Their contents was focussed more on lexical and syntactic issues of the programming languages.

The paper is organized as follows: first we give a description of our approach to design a checker for programming tasks in Haskell. After this, we describe the checker system and interaction possibilities seen from the students and instructors perspective. Then we shortly sketch the use and benefits of multiple choice questionnaires within the overall checker and teaching environment. This is followed by the summarized feedback of our students. We conclude the paper with a discussion and future work section.

2. THE CHECKER
2.1 Designing the checker in HASKELL
The initial design of the core functionality of the so called checker program has been done in Haskell. The checker’s task is to automate checking and preselection of students’ submissions for mandatory programming tasks. The Haskell implementation served as an ideal basis for communicating and clarifying the initial ideas within the group of people involved.

The design discussions centered around the possibilities for alternative solutions for typical programming tasks and around possible error cases. The following example serves to illustrate the relevant arguments.
Suppose the students are given the following task.¹

Implement a Haskell function `remDuplicates` that, when given a list, returns a list with all duplicate elements removed.

The instructor may have expected a solution like:

```haskell
-- direct implementation
remDuplicates [] = []
remDuplicates (x:xs) =
  if (elem x xs) then remDuplicates xs
  else x : remDuplicates xs
```

A student solution could as well be:

```haskell
-- indirect with utility function removeOccs
remDuplicatesB [] = []
remDuplicatesB (x:xs) =
  x : remDuplicatesB (removeOccs x xs)
```

Both solutions are valid, but they produce results that are different when a simple comparison with equality is used. Since the task description does not restrict the order of the elements remaining the resulting lists should be accepted as equal when they are permutations of another.

Based on these arguments, the initial prototype of the checker has been coded in Haskell as follows²:

```haskell
module Checker where

-- compare the lists of results element wise

testPairwise [] [] _ = True

testPairwise (x:xs) (y:ys) equiv =
  if (equiv x y) then testPairwise xs ys equiv
  else False

-- return True, if all pairs match under equiv,
-- otherwise return False

checkFoosOnData master totest list equiv =
  testPairwise (map master list) (map totest list) equiv
```

With this solution, the comparison between the alternative versions of `remDuplicates` yields the following results:

```
RemDups> remDuplicates [1,2,3,2,4,1]
[3,2,4,1]
RemDups> remDuplicatesB [1,2,3,2,4,1]
[1,2,3,4]
RemDups> checkFoosOnData remDuplicates remDuplicatesB

[[1,2,3], [1,2,3,3,2,1]] listEqualModPermutation True
RemDups> checkFoosOnData remDuplicates remDuplicatesB

[[1,2,3], [1,2,3,3,2,1]] (==) False
RemDups> checkFoosOnData remDuplicates remDuplicatesB

[[1,2,3]] (==) True
```

The examples serve to illustrate once more that for relevant results both the set of test data (`[[1,2,3], [1,2,3,3,2,1]]` vs. `[[1,2,3]]`) and the comparison function (`listEqualModPermutation` vs. `(==)`) have to be carefully selected.

### 2.2 Design rationale

Full analysis of the students’ code would be very attractive. But this goes far beyond what we wanted to achieve.

We wanted to automate only the preselection of students’ submissions into those that the students definitely had to work over again and into those that possibly were correct and only had to be double checked by the supervising assistant.

In order to achieve this goal, we proceeded along the following guidelines:

- The solution should be as general as possible and should at best allow to declaratively add new languages and their programming environment (e.g. interpreter, debugger, compiler, etc.).
- Functionality already available should be used rather than re-implemented. For example, lexical analysis of students’ programs should be performed by the interpreter/compiler and not by a newly coded component.
- An internationalized standard format such as XML should be used for data storage (programming tasks, students’ submissions, checker results, etc.) and application’s interfaces. This makes it possible to easily integrate the software into an existing system infrastructure.

---

¹The original task texts are in German, the translation is given here for convenience.
²For ease of integration into the course management system based on the open source Zope/Plone content management framework [3] [2] the production version of the checker is implemented in Python [1].
2.3 Interaction with the system

2.3.1 The student perspective

The students have to subscribe to the course management.
After this, they have their individual password protected accounts.

For submission of programming solutions, the student uses a web interface in his/her most favorite browser. S/he may either type in her/his solution or upload it from a file.

All submissions are stored in a database. The checker gets informed about new entries.

The first test of the checker is: Is the program free of syntax errors?

If not, the student is informed and prompted to re-submit an improved solution.

A syntactically correct program is then run within the programming environment (for Haskell we use runhugs) with the set of test data. If the program does not terminate within a certain time limit it is treated as possibly faulty and sent back to the student together with the hint to check for possible infinite loops or infinite recursions.

For a terminating program, the result data for the set of test data are compared with the corresponding results from the master solution. The test may involve the application of a comparison function. This is foreseen for cases where correct results are not uniquely determined but may vary. For example, resulting lists (see above) can contain the elements expected but in a variety of possible orders.

If there are deviations between the expected results and the students’ solution even modulo comparison function then the student gets informed about these differences and is again prompted to re-submit an improved solution.

2.3.2 The instructor’s perspective

To feed exercises into the system an instructor may either choose an exercise already in the repository or may add a new one to the collection. The following data and metadata are needed for a new programming task:

- a text explaining the task,
- a target programming language or languages,
- a function call (name, parameters),
- a function’s result type,
- a master solution,
- a set of test data, and
- compulsory: a comparison function.

An authoring tool allows the instructor to provide this information without the need for XML expertise. Instructors should be free to concentrate on the issues of designing tasks and e.g. choosing sets of test data. Test data should not only cover all relevant cases to be considered by the student and his/her program. In the case of failure, they should as well serve as stimuli for the student to re-think the task and his/her solution so far.

The checker is integrated into course management functionality. Relevant aspects of the interaction of the students with the system are recorded (e.g. number of trials, wrong and possibly correct solutions). Based on this, students have the possibility to always get an up-to-date record of their own performance in task fulfillment for the running course. The instructors can at any time get accumulated data about the performance of all participants or of subgroups. This information can be helpful to detect issues and topics that caused major problems for the students and therefore should be taken up once more in either the lecture or in the group exercises.

In addition a positive effect of electronic submissions is an increasing collection of student solutions. Enhanced with meta information (e.g. year and course) such a repository can be useful for lecturers or assistants. Maybe if an assistant is reusing a task from the last semester s/he can refer in the group exercise to a particularly original or to an exceptionally bad solution.

2.3.3 An example session

The snapshots of a student’s interaction with the checker (see figure 1 and 2) are intended to give a flavor of the actual usage of the system.

Figure 1: Submitting a solution for a programming task.

We suppose that the student has not read the text of the programming task remDuplicates carefully enough. S/he therefore assumed that only duplicates in immediate neighborhood should be removed. But s/he had not only a wrong
concept of the task in mind his/her solution of the (wrong)
task was faulty as well. This is the submitted attempt:

```haskell
remDuplicates [] = []
remDuplicates [x] = [x]
remDuplicates (x:y:xs) =
    if (x == y) then x : remDuplicates xs
    else x : remDuplicates (y:xs)
```

The checker is run with this submission. The student gets
feedback via e-mail (figure 3)5 or s/he may access her/his
account (figure 2).

Now the student should – hopefully – re-think his/her so-
lution. The expected outcome for the list \([1,2,3,3,2,1]\)
should help to detect the initial conceptual flaw: all, not only
immediately consecutive, duplicates have to be removed.
The analysis of the difference between the expected outcome
and the achieved outcome for \([1,2,2,2,3]\) should help to
recognize the error in coding the recursion in the submitted
version.

If this results in a submission that is accepted the student
receives a corresponding affirmative feedback, if not, s/he is
re-prompted again.

### 2.3.4 Induction proofs

In addition to solutions of programming tasks, the students
have to submit induction proofs about properties of Haskell
programs as well. For these submissions, the evaluation is
done by the assistants.

Their task is made somewhat easier because they get the
proofs presented in a layout that makes the inherent struc-
ture obvious. The student interface for induction proof sub-
missions is organized in such a way that the logical parts of
an induction proof (i.e. induction base, induction assump-
tion, induction step) are captured separately. This struc-
tured capture is helpful for the students as well.

We observed that the relative ease to prove program pro-
erties in Haskell is fascinating for many students but that
on the other hand many of them have problems with per-
forming induction proofs on their own, so any additional
structural guidance should be helpful. Figure 4 shows an
input mask6 for the following task for the students:

```
Prove by induction about length of \(xs\) that
the following holds:

\[
\text{map } f \ (xs \ ++ \ ys) = \text{map } f \ xs \ ++ \ \text{map } f \ ys
\]
```

For each step of the proof a separate input field is given in
order to support the students during there submission of the
solution.

### 2.3.5 Internationalization

So far, the overall checker system only supports German. To
increase the usability - especially for universities with many

---

5Please find an enlarged version of this image at http://lls.cs.uni-magdeburg.de/public/docs/haskelws/.  
6Please find an enlarged version of this image at http://lls.cs.uni-magdeburg.de/public/docs/haskelws/.
international students - it is our aim to support multiple languages.

For this purpose, the XML structures of programming tasks and induction proofs must be extended as well as the authoring tools and presentation applications.

As mentioned before, interaction with the students takes place within the Zope/Plone environment. Zope/Plone itself offers multilingual support and localization. So if programming tasks and induction proofs already contain text in different languages it is quite easy to present it through Zope/Plone’s language tools.

Based on the student’s preferred language, the resulting e-mails of the checker have to be localized as well. For this purpose, it is possible to import and use the Zope/Plone language files within the checker’s e-mail module.

2.4 Redesign: towards a generic solution

The basic cycle of the checker is quite generic (see figure 5).

- Take a student solution in a specific programming language.
- Use the language environment to test:
  - Is the program syntactically correct?
  - Is it terminating on the test data?
  - Is it producing results on the test data that – taking a comparison function into account if needed – are in accord with the results gained from applying the master solution on the test data set?
- Depending on the outcome inform the student directly (and the instructor indirectly).

The current version of the checker supports Haskell, Scheme, and CommonLisp. Re-design and re-implementation within the Zope/Plone environment is already in progress. It is intended to achieve a more generic level that allows to add additional programming languages (e.g. Python, Prolog, and Java) to the system by declaring the necessary language specific data in an XML structure that serves to configure the checker appropriately.

3. MULTIPLE CHOICE QUESTIONNAIRES

In addition to the support for programming tasks and induction proofs, we wanted to offer web based self tests in the style of multiple choice questionnaires to the students. Their contents was focussed more on lexical and syntactic issues of the programming languages (see figure 6).

The software to present, evaluate and summarize multiple choice questionnaires is based on an XML structure for the representation of questions and sets of answers. This XML structure has been designed in such a way that the instructor can concentrate on the test design, i.e. selection of questions and answer sets. Furthermore, s/he can choose between a variety of scoring schemes (e.g. with and without penalties for wrong answers) as well as the type of the questionnaire (multiple-answer vs. single-answer). The interaction with the student and the result evaluation are realized by a separate module that exploits the information from the XML schema.

In order to block students from simply taking the answer sets of a friend, each questionnaire is ‘individualized’ by randomly choosing possible questions and possible answer sets as well as ordering them randomly.

---

7 We have to focus on task texts and comments only, all other elements are not concerned.
8 Several tools exist for managing multilingual content and the Plone interface has been translated into over 30 languages. Since version 2.0 multilingual support and localization are an essential part of Plone.
9 Please find an enlarged version of this image at http://lls.cs.uni-magdeburg.de/public/docs/haskellws/.
10 Please find an enlarged version of this image at http://lls.cs.uni-magdeburg.de/public/docs/haskellws.
4. STUDENT FEEDBACK
To get feedback for our future work, we asked the students to fill out a questionnaire at the end of the course. The summarized results are, that:

- the majority of the students felt positive about the possibility to solve programming tasks at home and submit solutions online via a web browser.
- no appreciable technical problems appeared.
- some information given by the checker system were not assessed as helpful by the students (e.g. wrong line numbers in error messages, which resulted from wrapping the students solutions). Therefore: to ensure meaningful and informative feedback is one of the future tasks.
- by a majority of the students the current multiple choice questionnaires for Haskell were not regarded to be necessarily meaningful for learning control. Therefore, we have to consider the topics for multiple choice tests as well as to improve the automatic evaluation.

5. DISCUSSION AND FUTURE WORK
A major positive effect of the checker system has been that most students have really implemented and tested their solutions and not simply relied on sketchy pencil and paper approaches.

In order to avoid that students are simply copying their friend’s solution and submitting it, we have checked submissions for identical or nearly identical solutions (i.e. solutions substantially different from all previous submissions) whereas solutions classified as ‘(nearly) identical to already existing submission’ will receive only minimal scores. We hope that this will be a challenge for the students and will stimulate their creativity to exploit the multitude of possible ways to solve a given task that are available especially in Haskell.

6. ACKNOWLEDGMENTS
The work reported is performed under grant 0047M1/0002A from a program on ‘Multimedia for higher education‘ of the federal state of Sachsen-Anhalt, Germany.

We have to thank our colleagues Ilona Blümel and Sylke Krötzsch. They have pioneered the usage of the system in parallel to organising and running the group exercises for ‘Programming concepts and modelling‘ in the last winter semester and they have with full dedication initially populated the XML based repository with annotated programming and problem solving tasks.

In addition, we have to thank our students R. Bubke, T. Sommerfeld, S. Steinmeyer (Haskell checker team) and P. Kruse, A. Naujoks (multiple choice questionnaires) for many inspiring discussions and for their implementation work.

Accessibility
The system is publicly available (username 003366 and password m940618) at:
http://lls.cs.uni-magdeburg.de/lehre/ws0304/pkm/uebung/insert_solution

All images used in this paper can also be found (with higher resolution) at:
http://lls.cs.uni-magdeburg.de/public/docs/haskellws/.

7. REFERENCES