Computer Aided Exercising in Prolog and SML

Dávid Hanák, Tamás Benkő, Péter Hanák, and Péter Szeredi
{dhanak,benko,hanak,szeredi}@inf.bme.hu

Faculty of Electrical Engineering and Informatics,
Budapest University of Technology and Economics, Hungary

Abstract. This paper introduces a fully computerised, web-based, interactive system which enables students of Computer Science at the Budapest University of Technology and Economics to solve simple exercises in two declarative programming languages (Prolog and SML). This system works as part of a comprehensive teaching support environment that helps both students and teachers of the *Declarative Programming* course.

1 Introduction

In the *Declarative Programming* course we teach Standard ML and Prolog for 4^{th} semester students of Computer Science at the Budapest University of Technology and Economics. For demonstration and evaluation purposes we use the *SICStus Prolog* and the *Moscow SML* environments.

In the first three semesters students learn several imperative programming languages including Pascal, C and C++, they also get some basic training in x86 assembly, and later on they become familiar with Java and SQL. On the other hand, this single semester course is the only opportunity for them to learn about declarative thinking in the context of programming.

It is a commonplace that programming requires practice, but how can students be induced to improve their skills? One way is the laboratory exercise but, unfortunately, we run short both on human and technical resources. The number of students attending the course has increased from about 100 to more than 400 per year in the last eight years, while the staff consists "only" of two lecturers and two PhD students. To tackle the problem we developed several tools to facilitate the work of the lecturers, and an interactive web-based program incorporating an extensive exercise database which allows students to practise 24 hours a day. These pieces of software have recently been integrated into a comprehensive teaching support system, the Environment for Teachers and Students (ETS).

2 An Environment for Teachers and Students

This section presents a sketch of ETS, and describes the exercise system in detail.

Because of the high number of students, it is vitally important to have as many auxiliary tasks automated as possible, such as the evaluation of assignments and exercises. Supporting the lecturers by lightening the burden of administrative tasks is also a relevant question.

This recognition lead us first to the implementation of several independent programs handling these tasks, and finally to the integration of these pieces of software into a comprehensive teaching support system called ETS (a Hungarian acronym. The best English equivalent of this acronym is *Environment for Teachers and Students)*. The design of ETS has been laid down in [1] and the system is currently under development.

The ETS consists of loosely connected components linked to a common database. Some components are already in use, others are unfinished yet, and still there are some which only exist in our plans.

One component provides an interface to the database for users with various privileges. Another component is responsible for managing assignments: this includes reception of the programs, running the tests, evaluating the results and notifying the students. A third component is an interactive exercise system designed especially for the needs of programming courses. The initial version of this component was reported in [2].

The latter two components will be introduced in the rest of this section.

2.1 Managing Assignments

Assignments consist of several simple programming tasks and a single complex problem issued in each semester. Points received for the solutions add up to the examination scores.

Along with each task we provide a testbed, a submission script and a set of test cases. The testbed contains procedures/functions for parsing test cases and pretty-printing solutions, so that the students do not have to bother with this and can fully concentrate on the algorithm. A very similar testbed is used for testing the solutions on the server-side, too. The submission script—running on Unix-like systems—must be used to hand in the programs via e-mail. This way we ensure that the students are properly identified (the script requests that the sender name be selected from a list), and also that all submissions use the same attachment format. The server automatically evaluates the submitted program, stores the results in the database, and sends a verbose log to the student.

An HTML form is also available for students who—being unfamiliar with Unix systems—may have problems using the submission script.

2.2 The Exercise System

The exercise system offers several types of exercise problems to the students, receives and checks their solutions. In case of an erroneous solution it reports the errors and offers re-editing. The system also follows and registers the progress of students by making statistics.

At the design of the exercise system, we had the following objectives in mind:

- user-friendly interface with easy navigation;
- "fool-proof" error handling covering all possible errors, and providing informative error messages;

– protection of confident data and the system itself against malicious programs;
– easy addition of new exercises and exercise types by the administrators;
– reduced possibility for human mistakes by avoiding redundancy: checking the solutions directly by running the exercise, rather than by comparison to a reference solution;
– organising tasks into topics for easier traversal.

In order to avoid overwhelming the students with exercises they cannot solve, the problems are gradually made available by the system, following the course curriculum. The first exercises the students encounter are very simple and require only the most basic skills in both languages. Then—as the semester goes on—they assume deeper and deeper knowledge.

In the rest of the section, we describe those topics of the course material which are supported by the exercise system, and then we discuss some implementation aspects of the system.

Exercise Topics in Prolog

Standard prefix notation. First the students must get to know how Prolog parses expressions, what are the atoms, compounds and operators. This is best practised by letting the students themselves do the work of the Prolog parser: the task is to specify the *canonical form* of an expression. Example:

Q: `a(2+3*5,b(1-2,4))`
A: `a(+(2,*(3,5)),b(-(1,2),4))`

Lists are fundamental in all declarative languages because of their recursive structure that suits recursive algorithms well. Therefore it is a must to understand how they are built. The canonical form expresses the recursiveness well, but using syntactic sugars may hide it. In this exercise the students must find the canonical form of list expressions. Example:

Q: `[2+3,[1]|T]`
A: `.(+(2,3),.(.(1,[]),T))`

Unification. When the notion of canonical form is clear, attention must be paid to the unification of expressions containing variables. The students, given two expressions, must tell whether unification succeeds or fails, and in case of success what values the variables will assume. Example:

Q: `[3+Y|Z] = [X+4*5,X].`
A: success, `X = 3`, `Y = 4*5`, `Z = [3]`

Equational operations. The four types of equations: unification (`=`), identity (`==`), arithmetic evaluation (`is`) and arithmetic equality (`=:=`) can be quite confusing for someone just learning Prolog. These exercises help understand the main differences between these predicates. The task is again to tell whether the call succeeds, fails or *signals an error*, and in case of success what values the variables will assume. Examples:

Q: `X is +(1,2), X =:= 10//3.`
A: success, `X = 3`
Q: `X is 5//4, X == 5//4.`
A: failure

Control. The next step is to introduce control structures like the cut (`!`), the conditional (`-> ;`), and the negation (`\+`). The question is the same as above. Examples:

Q: `U is 21/6, (U < 3 -> X is 3-U ; X is U-3).`
A: success, `U = 3.5`, `X = 0.5`
Q: `X = 2, \+ \+ X = 1.`
A: failure

Backtracking is of course the most powerful feature of Prolog. Students have to understand how a nondeterministic program is executed, how clauses are tried, what happens at cuts and in conditionals. In this type of exercise a small set of predicates is given, and the students have to tell what solutions are enumerated in the variable(s) of a specified call. Example:

Q:
```
p(1, 1).  p(3, 1).  p(_, 2).

q([H|T], A) :- p(H, A).
q([H|T], A) :- p(T, A).

| ?- q([1,2,3,4], X).
```

A: `X = 1; X = 2; X = 2`

Programming. When the students can handle all of the above "basic ingredients", they are asked to write simple predicates. The problem is specified by giving the *head comment* of the predicate, and some examples. Example:

Q:
```
% adj(+L, ?Sum, ?A, ?B): A and B are adjacent elements
%   in the L list of numbers, and their sum is Sum.

| ?- adj([2,3,1,4], 5, A, B).
A = 2, B = 3 ? ;
A = 1, B = 4 ? ;
no
```

Built-in predicates. As soon as they can write simple predicates from the scratch, it is time to start using built-in and library predicates. This is again the usual success/failure/error type exercise. Example:

Q: `append(X, _, [1,2]), X = [_|_], !.`
A: success, `X = [1]`

Meta-logic predicates perform operations that require reasoning about the current instantiation of terms (such as `atom/1` and `ground/1`) or decomposing terms into their constituents (`functor/3`, `arg/3` and `=../2`). The question is the same as above. Examples:

Q: `functor(X, +, 2), X =.. [_,1,2|_].`
A: success, `X = 1+2`
Q: `X =.. [_,1,2|_], functor(X, +, 2).`
A: instantiation error

Exercise Topics in SML

Basic types and values. The first thing students learn about SML is that it has a strong type system. They find out about primitive data types (like `int` and `bool`), the corresponding value sets, simple operators and functions manipulating these data, and how these values can be coupled into *tuples*. In the first exercises they are given a tuple containing basic expressions and have to specify its type and value the SML interpreter would give. Example:

Q: `("pi"^"e", 4 mod 2, 1<>0)`
A: `("pie", 2, true) : string * int * bool`

Lists are also fundamental in SML, but their syntax is to some extent different from the syntax of Prolog lists. Students learn how lists can be constructed with the *cons* (`::`) operator, how to use syntactic sugar, and also learn about the *append* (`@`) operator. Exercises at this stage ask the same kind of question as above, only for lists. Example:

Q: `[2.1, 4.2] @ real 3 :: [abs(~1.7)]`
A: `[2.1, 4.2, 3.0, 1.7] : real list`

Understanding simple functions. The next step after learning about the basic data types is to come to know the most important SML construct, namely the notion of the function, and to learn about *pattern matching* and conditional constructs. Students also have to recognise (as yet monomorphic) function types. In these exercises we ask the type of a function or the return value of a function call. Examples:

Q: `fun f (c,s) = explode (s ^ str c)`
A: `f : char * string -> char list`
Q: `f` as above, `f (#"e", "pi")` = ?
A: `[#"p", #"i", #"e"] : char list`

Writing simple functions on your own stimulates constructive thinking already at this early stage. Students are given the specification and type of a function, and have to implement it. Example:

P:
```
(* sum xs = the sum of integers in ‘xs’
   sum : int list -> int                *)
sum [4,6,3] = 13
```

Lambda notation comes up as the canonical form of function definitions. These exercises ask either the type or the value of an expression containing a lambda function. Examples:

Q: `val f = fn (s,c) = str c ^ s`
A: `f : string * char -> string`
Q: `(fn x => x*x) 2`
A: `4 : int`

Polymorphism. Since students already know about lists and tuples, it is reasonable to teach them how they can handle such constructs in general, without knowing the type of their constituents in particular. Here we ask

- the type of a function with specified body; or
- a possible body with given type and head; or
- a function definition given its specification.

Examples:

Q: `fun f x ls = length ls > x`
A: `f : int -> 'a list -> bool`
Q: `f : 'a list -> 'a list * 'a`
`fun f (x::xs) =` ?
A: `fun f (x::xs) = (xs, x)`
P:
```
(* lgr (l,ls) = 'ls' is longer than 'l'
   lgr : int * 'a list -> bool          *)
```

Datatype declarations. The ability of constructing user defined data types (like binary trees) is a very strong feature of SML. The task here usually is to write a function operating on a predefined data type. Example:

P:
```
datatype ('a, 'b) union = A of 'a
                        | B of 'b
(* split xs = (as,bs) where 'as' is the list of all values A,
              'bs' is the list of all values B from 'xs'
   split : ('a, 'b) union list -> 'a list * 'b list        *)
```

Eager and lazy evaluation. Students learn quite early that the evaluation strategy of SML is eager, but there are a few exceptions when lazy evaluation is used. These are the logical operators (`andalso`, `orelse`, `if-then-else`) and the function definition, where the body is evaluated only on calling the function. The goal here is to call their attention to this feature. An expression is given with calls to `print` and `printVal`, and the output of the evaluation must be specified as the answer. Examples:

Q: `printVal 1 > 0 orelse printVal(null [1])`
A: `1`
Q: `(fn (x,y) => print x) ("app", printVal "le")`
A: `"le"app`

Partially applicable functions. If we know that lazy evaluation is available in SML, we might be interested in delaying some calculations, and bringing others forward. This can be solved with *curried functions*, which come in handy in lots of other situations as well. These exercises ask the type of partially applied functions. Examples:

Q: `fun f x y = Math.sqrt(x*x + real(y*y))`
`val g = f 4.0`
A: `f : real -> int -> real`
`g : int -> real`

Higher order functions. Eventually students learn about functions that take a function as an argument, like `map` and `foldl`. The exercises of this topic ask the type and/or value of expressions containing these functions. Example:

Q: `foldl op o chr`
A: `(char -> char) list -> int -> char`
Q: `map (fn i => i div 2) [3,4,6,9]`
A: `[1,2,3,4] : int list`

About the Implementation. We have defined and use the following notions:

- The exercise *scheme* gives the means of how a task is presented, in what form the answer is expected, how it is checked, and how the result is reported back to the user—naturally between the limits imposed by the exercise system itself and the capabilities of HTML forms.
- The *category* defines the section of the course material the current task belongs to.
- Categories may form *category groups* to help navigation: instead of selecting categories from a long flat list, they are presented in a hierarchy. One category may belong to several groups.

Categories can be formed by refining and further partitioning the topics discussed so far. The topics themselves can serve as category groups. Here is a list of the most important schemes:

Prolog schemes

Standard prefix notation: give the canonical form of an expression.

Success/failure/error: give the result of a call, in case of success also determine the value of a specific variable.

All solutions: enumerate (in proper order) all solutions of a goal, as returned in a specified variable.

Programming: write a predicate satisfying a given specification.

SML schemes

Type: determine the type of a declaration (value or function).

Value: determine the simplest form of the value of an expression.

Function body: determine the body of a function if the head and the type is given.

Type declaration: define a data type satisfying a specification.
Programming: write a function conforming to a given specification.

Checking the exercises is not always simple, and calls for tricks to handle every possible error in a clever way. First, the solution must pass a syntax test, and only then can we move on to the semantic test, which in turn may consist of several rounds. Often the solution has to be tested against minimal requirements, and then checked whether it is not over-specific.

For example, to check that the type of an SML declaration is really what the student stated and not more generic, we wrap the declaration and the type into an SML structure.

```
structure expr :
  sig val y : correct type end =
struct
  declaration of x
  val y : type guess = x
end
```

This way the message of the SML interpreter is different if the type is altogether wrong, or if it is just over-specific.

To check Prolog standard prefix notations, an appropriate DCG parser has been included in the system.

3 Conclusions

We described that the increasing number of students presents a problem, which we try to solve by introducing automated tools in exercising, evaluation and administrative tasks. These tools have been integrated into a comprehensive teaching support system called ETS. We briefly presented the main components of ETS, and discussed the exercise system in greater length.

Acknowledgement. Thanks are due to all students who have helped us in the implementation of the ETS and its ancestors; especially to Mr. Lukács Tamás Berki and Mr. András György Békés for their work on the exercise system.

References

1. Dávid Hanák: *Computer Support for Declarative Programming Courses* (in Hungarian), 2001, MSc Thesis, see `<http://www.inf.bme.hu/~dp/publ/ets-hd.pdf>`. See also `<http://dp.iit.bme.hu:4321/>`
2. András György Békés, Lukács Tamás Berki: *A Web-based Exercise System for Programming Languages* (in Hungarian), 2001, Students' Conference, Budapest, Hungary
3. Péter Hanák, Péter Szeredi, Tamás Benkő, Dávid Hanák: *"Yourself, my lord, if no servants around" – A Web-based Intelligent Tutoring System* (in Hungarian), Networkshop 2001, Sopron, Hungary, see `<http://www.inf.bme.hu/~dp/publ/magadUram.pdf>`.