A Pragmatic Approach to Pre-Testing Prolog Programs

Christoph Beierle, Marija Kulaš, Manfred Widera

Praktische Informatik VIII - Wissensbasierte Systeme
Fachbereich Informatik, FernUniversität in Hagen
58084 Hagen, Germany
{beierle | marija.kulas | manfred.widera}@fernuni-hagen.de

Abstract. We present an overview on the AT(x) approach which is capable of automatically analyzing programs with respect to given tests and a reference solution. In the context of small homework assignments with precisely describable tasks, AT(P), a Prolog instance of the general AT(x) framework, is able to find many of the errors usually made by students and to communicate them in a manner understandable for beginners in Prolog programming. The system is being used in distance education where direct communication between students and tutors is most of the time not possible.

1 Introduction

In distance learning and education, direct interaction between students and tutors is (most of the time) not possible. While communication via phone, e-mail, or newsgroups helps, there is still need for more direct help in problem solving situations like programming. In this context, intelligent tutoring systems have been proposed to support learning situations as they occur in distance education. A related area is tool support for homework assignments. In this paper, we will present a pragmatic approach to the automatic revision of homework assignments in programming language courses. In particular, we show how with the AT(P) system, exercises in Prolog can be automatically analyzed and tested so that automatically generated feedback can be given to the student. We will present an overview on AT(P) which is a Prolog instance of our more general AT(x) framework. Whereas AT(x) is introduced in [BKW03], this paper provides a more detailed description of the AT(P) functionalities.

2 WebAssign and AT(x)

The AT(x) framework is designed to be used in combination with WebAssign, a general system for assignments and assessment of exercises for courses which

The research reported here was partially supported by the *Innovationsfond "Lernraum Virtuelle Universität" (LVU).*

was developed by H.-W. Six and his group [BHSV99,Web03]. It provides support with web-based interfaces for all activities occurring in the assignment process, e.g. for the activities of the author of a task, a student solving it, and a corrector correcting and grading the submitted solution. In particular, it enables tasks with automatic test facilities and manual assessment, scoring and annotation. WebAssign is integrated in the Virtual University system of the FernUniversität Hagen [LVU03].

From the students' point of view, WebAssign provides access to the tasks to be solved by the students. A student can work out his solution and submit it to WebAssign. Here, two different submission modes are distinguished. In the so-called *pre-test mode*, the submission is only preliminary. In pre-test mode, automatic analyses or tests are carried out to give feedback to the student. The student can then modify and correct his solution, and he can use the pre-test mode again until he is satisfied with his solution. Eventually, he submits his solution in *final assessment mode* after which the assessment of the submitted solution is done, either manually or automatically, or by a combination of both.

While WebAssign has built-in components for automatic handling of easy-to-correct tasks like multiple-choice questions, this is not the case for more complex tasks like programming exercises. Here, specific correction modules are needed. The AT(x) framework aims to analyze solutions to programming exercises and is such a system that can be used as an automatic correction module for WebAssign. Its main purpose is to serve as an automatic test and analysis facility in pre-test mode.

Instances of the AT(x) framework have a task database that contains an entry for each task. When a student submits a solution, AT(x) gets an assignment number identifying the task to be solved and a submitted program written to solve the task via WebAssign's communication components. Further information identifying the submitting student is also available, but its use is not discussed here. Taking the above data as input, AT(x) analyzes the submitted program. Again via WebAssign, the results of its analysis are sent as feedback to the student (cf. Fig. 1).

3 An Example Session

Before we go into the description of the individual components of the AT(x) system, we want to show an example execution for a Prolog homework task. The task is described as follows:

> Let N and M be natural numbers with $N \leq M$. Define a predicate *between/3* such that a query `between(X, N, M)` is true if `N` is less or equal to `M` and `X` is a number between `N` and `M`.

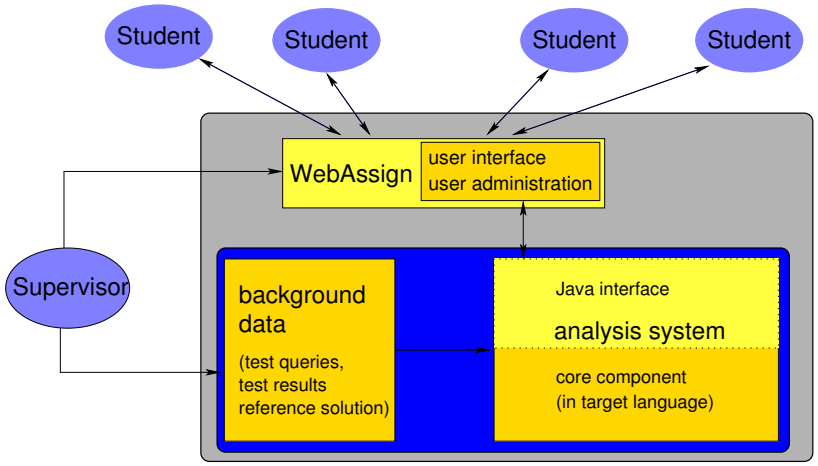

Fig. 1. Structure of AT(x)

Let us assume that the following program is submitted:

```
between(X, N, M) :- var(X), integer(N),
                    integer(M), N =< M,
                    gen_list(N, M, X).

gen_list(N, N, [N|[]]) :- !.
gen_list(N, M, [N|R]) :- L is N + 1, gen_list(L, M, R).
```

Then the system's output is the following:

```
The following query failed, though it should succeed:
between(10,10,20)
-------------------------------------------------
Wrong solutions were generated for the following query:
between(A,100,102)

The wrong solutions for this query are listed below:
between([100,101,102],100,102)
-------------------------------------------------
Solutions were overlooked for the following query:
between(A,100,102)

The overlooked solutions for this query are listed below:
between(100,100,102)
between(101,100,102)
```

```
between(102,100,102)
```

One interesting aspect of the AT(x) framework is the following: the system is designed to perform a large number of tests. In the generated report, however, it filters some of the detected errors for presentation. Several different filters generating reports of different precision and length are available. In the example above, a single representative for each kind of detected error was selected.

4 Structure of the AT(x) Framework

The AT(x) framework consists of combining different tools. Interfaces to different user groups (especially students and supervisors) have to be provided via WebAssign. The design decisions caused by this situation are described in this section.

4.1 Components of the AT(x) System

AT(x) is divided into two main components: the main work is done by the analysis component. Especially in functional and logic programming, the used languages are well suited for handling programs as data. The analysis components of AT(P) (and also of AT(S), an instance of AT(x) for the functional programming language Scheme cf. [BKW03]) is therefore implemented in the target language (i.e. the language the programs to be tested are written in).

A further component implemented in Java serves as an interface between this analysis component and WebAssign. The reason for using such an interface component is its reusability and its easy implementation in Java. The WebAssign interface is based on Corba communication. A framework for WebAssign clients implementing an analysis component is given by an abstract Java class. Instead of implementing an appropriate Corba client for each of the AT(x) instances in the individual target languages independently, the presented approach contains a reusable interface component implemented in Java (that makes use of the existing abstract class) and a very simple interface to the analysis component.

4.2 The Analysis Components

The individual analysis components are the main parts of the AT(x) instances. They perform a number of tests on the students' programs and generate appropriate error messages. The performed tests and the detectable error types of AT(P) are discussed in Sec. 5 and 6. Here, we concentrate on the (quite simple) interface of this component.

The analysis component of each AT(x) instance expects to read an exercise identifier (used to access the corresponding information on the task to solve) and a student's program from the standard input stream. It returns its messages, each as a line of text, at the component's standard output stream. These lines of text contain an error number and some data fields containing additional error descriptions separated by a unique identifier. The number and types of the additional data fields is fixed for each error number.

4.3 Function and Implementation of the Interface Component

On the one hand, WebAssign provides a communication interface based on Corba to the analysis components. On the other hand, the analysis components used in AT(x) use a simple interface with textual communication via the stdin and stdout streams of the analysis process. We therefore use an interface program connecting the analysis component of AT(x) and WebAssign which performs the following tasks:

- Starting the analysis system and providing an exercise identifier and the student's program.
- Reading the error messages from the analysis component.
- Selecting some of the messages for presentation.
- Preparing the selected messages for presentation.

The interface component starts the analysis system (via the Java class *Runtime*) and writes the needed information into its standard input stream (which is available by the Java process via standard classes). Afterwards, it reads the message lines from the standard output stream of the analysis system, parses the individual messages and stores them into an internal representation.

During the implementation of the system it turned out that especially SICStus Prolog generates a number of messages at the stderr stream, e.g. when loading modules. These messages can block the Prolog process when the stream buffer is not cleared. Our Java interface component is therefore able to consume the data from the stderr stream of the controlled process without actually using them.

For presenting errors to the student, each error number is connected to a text template that gives a description of this kind of error. An error message is generated by instantiating the template of an error with the data fields provided by the analysis component together with the error number. The resulting text parts for the individual errors are concatenated and transferred to WebAssign as one piece of HTML text.

For using this system in education it turns out that presenting all detected errors at once is not the best action in every case. The interface component therefore has the capability of selecting certain messages for output according to one of the following strategies:

- Only one error is presented. This is especially useful in beginners courses, since a beginner in programming should not get confused and demotivated by a large number of error messages. He can rather concentrate on one message and may receive further messages when restarting the analysis with the corrected program.
- For every type of error occurring in the list of errors only one example is selected for output. This strategy provides more information at once to experienced users. A better overview over the pathological program behaviour is given, because all different error types are described, each with one representative. This may result in fewer iterations of the cycle consisting of program correction and analysis. The strategy, however, still hides the full

set of all test cases from the student and therefore prevents fine tuning a program according to the performed tests.
– All detected errors are presented at once. This provides the complete overview over the program errors and is especially useful when the program correction is done offline. In order to prevent fine tuning of a program according to the performed tests, students should be aware that in final assessment mode additional tests not present in the pre-test mode will be applied.

4.4 Global Security Issues

Security is an issue that is common to all instances of AT(x). It is therefore addressed by the framework rather than by every individual instance. In [Wid04] it is shown how authentification problems and denial of service attacks are dealt with and by which means malicious code in submitted programs can be detected. Essentially, WebAssign already provides a filter ruling out denial of service and unauthorized access. Malicious code in students' solutions (e.g. file access, ...) is prevented by the known UNIX security mechanisms. A sandbox approach is possible, but did not prove necessary so far.

5 Requirements for the Analysis Components

The heart of the AT(x) system is given by the individual analysis components for the different programming languages. The intended use in testing homework assignments rather than arbitrary programs implies some important properties of the analysis components discussed here: it can rely on the availability of a detailed specification of the homework tasks, it must be robust against non terminating input programs and runtime errors, and it must generate reliable output understandable for beginners.

The description for each homework task consists of the following parts:

– A textual description of the task. (This is essentially used in preparation of the homework assignment, but not in the testing task itself.)
– A set of test cases for the task.
– Specifications of program properties and of the generated solutions. (This applies especially for declarative languages like Prolog.)
– A reference solution. (This is a program which is assumed to be a correct solution to the homework task and which can be used to judge the correctness of the students' solutions.)

This part of input is called the *static input* to the analysis component, because it usually remains unchanged between the individual test sessions. A call to the analysis system contains an additional *dynamic input* which consists of a unique identifier for the homework task (used to access the appropriate set of static input) and a program to be tested.

Now we want to discuss the requirements on the behaviour of the analysis system in more detail. Concretizing the requirement of reliable output we want

our analysis component to return an error only if such an error really exists. Where this is not possible (especially when non termination is assumed), the restricted confidence should clearly be communicated to the student, e.g. by marking the returned message as a *warning* instead of an *error*. For warnings the system should describe an additional task to be performed by the student in order to discriminate errors from false messages.

Runtime errors of every kind must be caught without affecting the whole system. For instance, if executing the student's program causes a runtime error, this should not corrupt the behaviour of the other components. Towards this end, our AT(P) and AT(S) implementations exploit the hooks of user-defined error handlers provided by SICStus Prolog and MzScheme, respectively. An occurring runtime error is reported to the student, and no further testing is done, because the system's state is no longer reliable.

For ensuring termination of the testing process, infinite loops in the tested program must also be detected and interrupted. As the question whether an arbitrary program terminates is undecidable in general, we chose an approximation that is easy to implement and guarantees every infinite loop to be detected: a threshold for the maximal number of function calls[1] (either counted independently for each function or accumulated over all functions in the program) is introduced and the program execution is aborted whenever this threshold is exceeded. (In the presence of further looping constructs like *while*-loops in imperative programming, a refined termination control is necessary.) As homework assignments are usually small tasks, it is possible to estimate the maximal number of needed function calls and to choose the threshold sufficiently. The report to the student must, however, clearly state the restricted confidence on the detected non-termination.

6 Analysis of Prolog Programs

Annotation Tests Due to the declarative character of Prolog, a variety of tests can be performed on Prolog programs. Certain main properties of a program can be tested by annotations. Our system AT(P) is based on the TSP-approach of H. Neumann [Neu98] where several kinds of Prolog annotations are proposed, together with an algorithm for their validation with respect to a given student's program and a reference program. One general kind of annotation is a positive/negative annotation. Such an annotation consists of a test query, a flag whether this query should succeed or fail and a description of the property that is violated if the query does not behave as expected. This description is reported to the student together with the query and the intended result.

Example 1. Consider the predicate *between*/3 that succeeds for every call of the form $between(X, Y, Z)$ such that $X, Y, Z \in \mathbb{N}$ and $X \leq Z \leq Y$.

A possible error of a student's program is to allow for too large values Z. We can detect and explain that by an annotation test where

[1] in case of Prolog: predicate calls

– the test query is set to "$between(5, 10, 11)$",
– the success flag is set to expected failure,
– the error explanation text is "The third argument must not be greater than the second one.".

For instance, if this test fails, the system will generate the following output:

> The third argument must not be greater than the second one. The goal `between(5, 10, 11)` succeeded though it was expected to fail.

Mode Tests Mode tests form the class of tests that is probably performed most often by AT(P). Its aim is to check whether the given program (or more precisely, the currently checked predicate in this program) behaves correctly for all intended modes (i.e. combinations of input and output instantiations of the predicate). Performing a mode test consists of the following steps:

1. Generate test queries for all intended modes of the tested predicate.
2. For each generated query perform the following steps independently:
   (a) Evaluate the query with respect to the student's program and collect all generated results.
   (b) Evaluate the query with respect to the reference program and collect all generated results.
   (c) Search for evidence for errors in the generated result lists.

The generation of test queries uses a list of fully instantiated terms that should be provable by the tested predicate and a list of modes the predicate should be applicable with. In the mode list the individual parameter positions are marked as input or output parameter as usual: + denotes an input parameter that must be instantiated, − denotes an output parameter that must not be instantiated, and parameters marked with ? may be either way.

Example 2. Consider the well-known $append/3$ predicate as the tested predicate.

```
append([], L, L).
append([H|R1], L2, [H|R]) :- append(R1, L2, R).
```

Some example terms (provable goals) for this predicate are

$$append([], [a, b], [a, b]), append([a], [], [a]), append([a], [b, c], [a, b, c]).$$

Let the list of modes of $append/3$ be the following.[2]

$$append(?L1, ?L2, +L3), append(+L1, +L2, ?L3)$$

[2] In contrast to the usual mode declaration $append(?Prefix, ?Suffix, ?Combined)$ found e.g. in [SIC01], here we are not interested in the capability of $append/3$ to guess list entries when they are not completely given as e.g. in the goal $append(L1, [a, b], L)$.

For the first example term and the first mode declaration we get the following list of test queries:

$$append(L1, L2, [a, b]), \quad append(L1, [a, b], [a, b]),$$
$$append([], L2, [a, b]), \quad append([], [a, b], [a, b]).$$

The other example terms and mode declarations are processed analogously.

Wrong and Missing Solutions Another part of the test procedure consists of the application of the student program to test queries and checking the resulting substitutions with the reference program. The following steps of comparison are performed:

- Solutions of a student's program are reported as *wrong* solutions if they are falsified by the reference program (i.e. if the query given by the solution fails in the reference program).
- A solution of the reference program is reported as *missing* if it is not subsumed by some solution of the student's program. (It is not sufficient for the student's program to *accept* every solution generated by the reference program. The student's program must rather be able to *generate* all these solutions.)
- For some of the test queries the number of expected solutions can be given (*completeness annotation*), and the number of solutions generated by the student's program is compared with this specification.

Example 3. Consider the task of implementing a predicate *perm*/2 that is fulfilled if both arguments are lists which are permutations of each other. Let the analyzed test query be `perm([a,b,c], L)`. Let further the programs generate the following instantiations for L:

	considered values
analyzed program	[a,b,c], [a,c,b], [b,a,c], [b,c,a], [c,a,b], [c,b,a]
reference solution	[c,b,a], [c,a,b], [b,c,a], [b,a,c], [a,c,b], [a,b,c]

Because of comparing of the solution sequences as *sets*, the system can infer the correctness of the analyzed program with respect to this query.

In case of an infinite number of solutions just a prefix is generated. The system is still applicable to those tasks with infinite solution set if a natural order on the solutions exists and therefore the generated solutions of the student's program and the reference program match. An example of a problem with natural order is the generation of all prime numbers. In contrast, there is no single natural order for generating all words over the alphabet $\Sigma = \{$`@,#,$`$\}$.

Redundant Solutions Redundant solutions, i.e. solutions erroneously occurring several times in the sequence of solutions, are detected by an algorithm

that interprets *every* repetition of a solution as an unintended one. If repeated solutions are intended by the problem, these messages can be filtered out later.

This procedure turned out to be sufficient since in our context of homework assignments the processed solution sequences are usually quite small.

Supervising Termination and Runtime Errors The central part of the Prolog analysis component is an original backtracking analyst [Neu98] that evaluates a test query with respect to a program and an expected number of answers, creating a (partial) list of answers and a status report. A query shall be evaluated two times, once with respect to the student's program and once with respect to the reference program. Both programs are held in memory in parallel using different modules.

During the evaluation of a query in a module, its termination behaviour is assessed by a meta-interpreter as follows. Goals for predicates defined in the module are resolved, whereas goals for built-ins or imported predicates are passed to the runtime system using timed-out *call/1*. The interpreter counts the number of resolutions and runtime calls. Upon exceeding the threshold, the current evaluation of the query is aborted and evidence for an infinite loop is reported.

The class of runtime errors contains all errors that are detected by the runtime system and cause the immediate termination of the computation (unless they are caught and processed as in our system). Runtime errors in Prolog programs contain among others

- existence errors (e.g. a non-existing predicate was called)
- instantiation errors (e.g. performing a mathematical computation with uninstantiated arguments)
- resource errors (e.g. no more memory).

Test evaluations performed by AT(P) are supervised and runtime errors are caught. If a runtime error occurs, the error message is passed to the student via WebAssign and the remaining tests are canceled. This cancellation avoids imprecise results for further tests caused by side-effects of the runtime error.

7 Implementation and Experiences

Both AT(P) and AT(S) are fully implemented and operational. The analysis components run under the Solaris 7 operating system and, via their respective Java interface components, serve as clients for WebAssign.

During the summer semester 2003, AT(P) was used in the framework of a course on deduction and inference systems at the FernUniversität Hagen, and both AT(P) and AT(S) are currently being used for a course on logic and functional programming. Part of all programming exercises in the course are supported by the system, but not yet all of them. Thus, since currently the system is available just for selected homework tasks, using the system means sending in homeworks on two different ways (WebAssign for the selected available tasks,

"plain paper" sent in by mail or e-mail for the remaining tasks). Nevertheless, two thirds of the students chose to use the system. Feedback from the students was positive in general, mentioning both, a better motivation to solve the tasks, and better insight in the new programming paradigm.

8 Related work

In the area of testing and analysis of Prolog programs there have been many proposals, ranging from theorem proving (eg. [Stä98]) to various forms of debugging (eg. [Duc99]) and systematic testing. Here we shall only refer to proposals with a strong declarative bias, i. e. using logical assertions (or annotations) for representing and/or validating program properties. The proposals differ along several axes: static or run-time analysis, restricting the target language or not, expressiveness of the annotation language.

Some authors restrict the target language by throwing out "impure" predicates. LPTP [Stä98] is an interactive theorem prover for a pure (no `cut`, `var` or `assert`) subset of Prolog. LPTP's language is a first-order logic, enriched with connectives for success, failure and universal termination. GUPU ([Neu97], [NK02]) is a teaching environment for a pure subset of Prolog. Before a student is allowed to feed in some clauses for a predicate, a partial specification (as a set of annotations) must be supplied. In GUPU the tutor supplies a reference solution as well [NK02]. GUPU's annotation language can express examples, counter-examples and termination statements.

Yet other authors take the challenge of the "full" or standard Prolog language. The first approach to validation of full Prolog seems to be the ADVICE package [O'K84]. It was followed by a theoretical work [DM88] on static validation. Some current practical approaches, performing run-time validation, include the language of annotations of the CIAO-Prolog [HPB99] and the NOPE system of annotations [Kul00]. The TSP system [Neu98], at the heart of our AT(P), performs run-time validation of full Prolog, and its annotation language can express examples, counter-examples, modi and numerical constraints on the number of computed answers.

Whereas e.g. LPTP is a powerful theorem prover and GUPU is a fully integrated teaching environment, we would like to stress the fact that the aim of AT(P) is a different one. It is an approach to support Prolog programming in distance education for beginners in logic programming. In such a context, the AT(P) assumption to have a correct reference solution available and to check the student's program with respect to this reference solution (rather than, say, with respect to a program-independent specification), seems to be justified. Furthermore, our particular system requirements of a completely WWW based system without intensive interaction characteristics on the one hand and the requirements for robustness with regard to program errors, runtime-errors, non-termination, etc. on the other hand, led to the pragmatic approach of coupling the analysis component AT(P) to WebAssign, thereby reusing WebAssign's communication and administration facilities.

9 Conclusions and Further Work

We have presented a brief overview on the AT(x) approach which is capable of automatically analyzing programs with respect to given tests and a reference solution. In the framework of small homework assignments with precisely describable tasks, AT(P) is able to find many of the errors usually made by students and to communicate them in a manner understandable for beginners in Prolog programming (in contrast to the error messages of most compilers).

There are other programming aspects that are not covered by AT(P). Examples are the layout of Prolog code, use of "imperative" programming style, etc. While there are systems dealing with such aspects (e.g. enforcing a particular layout discipline), in our AT(P) approach they are currently handled by a human corrector in the final assessement mode. Whereas it should not be too difficult to extend AT(P) in this direction, our priority in the design of AT(P) was the focus on program correctness by fully automated pre-testing of Prolog programming assignments.

Acknowledgements: The basis for the analysis component of AT(P) is taken from the TSP system, which was designed and implemented by Holger Neumann [Neu98]. TSP offers a variety of different tests and turned out to be extremely stable. We also thank the anonymous referees for helpful comments.

References

[BHSV99] J. Brunsmann, A. Homrighausen, H.-W. Six, and J. Voss. Assignments in a Virtual University – The WebAssign-System. In *Proc. 19th World Conference on Open Learning and Distance Education*, Vienna, Austria, June 1999.

[BKW03] C. Beierle, M. Kulaš, and M. Widera. Automatic analysis of programming assignments. In A. Bode, J. Desel, S.Ratmayer, and M. Wessner, editors, *DeLFI 2003. Proceedings der 1. e-Learning Fachtagung Informatik*, volume P-37 of *Lecture Notes in Informatics (LNI)*, Bonn, 2003. Köllen Verlag.

[DM88] W. Drabent and J. Małuszyński. Inductive assertion method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

[Duc99] M. Ducasse. Opium: An extendable trace analyser for prolog. *J. of Logic Programming*, 39:177–223, 1999.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In K. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, 1999.

[Kul00] M. Kulaš. Annotations for Prolog – A concept and runtime handling. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation. Selected Papers of the 9th Int. Workshop (LOPSTR'99), Venezia*, volume 1817 of *LNCS*, pages 234–254. Springer-Verlag, 2000.

[LVU03] Homepage LVU, Fernuniversität Hagen, `http://www.fernuni-hagen.de/LVU/`. 2003.

[Neu97] U. Neumerkel. A programming course for declarative programming with Prolog. http://www.complang.tuwien.ac.at/ulrich/gupu/material/1997-gupu.ps.gz, 1997.

[Neu98] H. Neumann. Automatisierung des Testens von Zusicherungen für Prolog-Programme. Diplomarbeit, FernUniversität Hagen, 1998.

[NK02] U. Neumerkel and S. Kral. Declarative program development in Prolog with GUPU. In *Proc. of the 12th Internat. Workshop on Logic Programming Environments (WLPE'02), Copenhagen*, pages 77–86, 2002.

[O'K84] Richard A. O'Keefe. *advice.pl*. 1984. Interlisp-like advice package.

[SIC01] *Swedish Institute of Computer Science. SICStus Prolog User's Manual*, April 2001. Release 3.8.6.

[Stä98] Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *J. of Logic Programming*, 36(3):241–269, 1998. Source distribution http://www.inf.ethz.ch/~staerk/lptp.html.

[Web03] Homepage WebAssign. `http://www-pi3.fernuni-hagen.de/WebAssign/`. 2003.

[Wid04] M. Widera. Testing scheme programming assignments automatically. In *Trends in Functional Programming*. Intellect, 2004. (to appear).