Proceedings of the 1st International Workshop on

Teaching Logic Programming
TeachLP 2004

September 8-9 2004, Saint Malo

M. Ducassé, U. Nilsson, D. Seipel (Eds)

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

Proceedings of the First International Workshop on Teaching Logic Programming: TeachLP 2004 Saint Malo, September 8–9, 2004

M. Ducassé, U. Nilsson, D. Seipel (Editors)

Preface

Following the panel discussion at the International Conference on Logic Programming 2003 in Mumbai, India, the first international workshop on Teaching Logic Programming, TeachLP 2004, was held in Saint Malo, France, on 8–9 September 2004. The meeting ran as a workshop in conjunction with the 2004 International Conference on Logic Programming, held on September 6–10, 2004.

Logic Programming (LP) and Constraint Logic Programming (CLP) are powerful programming paradigms, but hard to learn without sufficient assistance. To further spread the technology it should be taught to a broader range of computer science students. The aim of the workshop was to investigate what is currently taught and how; what should be taught and why.

Suggested topics for submissions included:

- teaching on different aspects of (C)LP
  - fundamentals of logic programming,
  - software engineering techniques for Prolog,
  - debugging techniques for Prolog,
  - practical applications,
- teaching special aspects of (C)LP, such as control structures and meta predicates,
- teaching Prolog/(C)LP in a few hours for CS students,
- tools for teaching,
  - tools for teaching (C)LP,
  - using (C)LP technology for e-learning systems,
- relation to other fields and paradigms,
- applying (C)LP in other courses,
- (C)LP and the ACM curriculum,
- (C)LP for the masses or for experts.

We would like to thank all the authors who have submitted a paper, and all colleagues who have served as reviewers in the program committee for their contributions to the success of the workshop. Many thanks also to the organizers of

the International Conference on Logic Programming ICLP'2004 for the scheduling and the local arrangements. Finally, we acknowledge the support of Linköping University Electronic Press for publishing the workshop proceedings.

Saint Malo, September 2004
Mirelle Ducassé, Ulf Nilsson, Dietmar Seipel

Organizing committee

Mireille Ducassé (IRISA/INSA de Rennes, France)
Ulf Nilsson (Linköpings Univ., Linköping, Sweden)
Dietmar Seipel (Univ. Würzburg, Germany, PC chair)

Program committee

Christoph Beierle (Fern-Uni Hagen, Germany),
Manuel Carro (Technical Univ. of Madrid, Spain),
Mireille Ducassé (IRISA/INSA de Rennes, France),
Ulrich Geske (Fraunhofer First Berlin, Germany),
Gopal Gupta (Univ. of Texas at Dallas, USA),
Michael Hanus (CAU Kiel, Germany),
Ulrich Neumerkel (TU Vienna, Austria),
Ulf Nilsson (Linköpings Univ., Sweden),
Enrico Pontelli (NMSU, Las Cruces, USA),
Dietmar Seipel (Univ. Würzburg, Germany),
Kazunori Ueda (Waseda University, Tokyo, Japan).

Contents

An On-line Course on Constraint Programming

Christine Solnon
LIRIS CNRS FRE 2672, Nautibus, Université Lyon I
43 Bd du 11 novembre, F-69 622 Villeurbanne cedex
christine.solnon@liris.cnrs.fr

Abstract

This paper describes an on-line course on constraint programming. This course is dedicated to students of the "e-miage" formation, which is a french remote formation to "Information Systems for Companies Management". This course is available (in french) at
http://www710.univ-lyon1.fr/∼csolnon/Site-PPC/e-miage-ppc-som.htm

1 Introduction

The MIAGE (Méthodes Informatiques Appliquées à la Gestion des Entreprises) is a popular french formation on "Information Systems for Companies Management" which is delivered in twenty french universities. This formation lasts for three years and delivers a "Maîtrise" diploma, nearly corresponding to a Master's degree.

The e-miage [4] is an on-line version of the MIAGE formation: students of the e-miage can earn their degree via the Internet. This remote formation aims at reaching new trainees who cannot follow traditional courses, either because they cannot physically attend them (foreign or handicapped students), or because they are professionals that are continuing education during their release time. Another goal is to improve the actual learning conditions of basic education students by providing them with complete courses and training exercises available via Internet.

E-learning allows students to earn their degrees with maximum efficiency and flexibility, without commuting nor schedule conflicts: students study via Internet whenever and wherever they choose. However, remote formation also raises drawbacks, and students may feel isolated, or get stucked on difficult points. To overcome these drawbacks, student progresses are followed by tutors: a tutor is a professor of the university from which the student is registered; he answers

student's questions via e-mail and evaluates his knowledge at the end of each course.

The e-miage formation is composed of fifty course units, each course unit roughly corresponding to fourty learning hours and to three ECTS (European Credit Transfer System). One of these course units is entitled "Artificial Intelligence" and is composed of 36 working sessions of one hour: 2 introductory sessions about artificial intelligence, 8 sessions about logic, 5 about Prolog, 7 about constraint programming, 4 about ontologies, 2 about problem solving, 3 about machine learning, and 5 about expert systems. There is no predefinite order for studying these lessons. However, some sessions may be required for studying other sessions. In particular, logic must be studied before Prolog, and Prolog before constraint programming.

This paper describes the sessions dedicated to constraint programming. We first discuss instructional objectives, and the reasons that guided the choice of the programming language used to illustrate this course. We then describe the content of each session. We conclude on a first feedback on this course, and on some related sources of information.

2 Instructional objectives and choices

This course does not aim at training experts of constraint programming, but is an introduction to this field: the goal is to train students to use a constraint programming language to solve problems. By means of competences, one can summarize our objectives by the 4 following points:

- knowing what is a constraint satisfaction problem (CSP),
- being able to model a problem as a CSP,
- knowing how a constraint solver works, and
- being able to use a constraint programming language to solve a CSP.

These objectives must be achieved within 7 training sessions of one hour so that we have limited this course to the very basic concepts: we mainly deal with basic constraint satisfaction problems, and only briefly introduce their different extensions, such as Max-CSP or Valued-CSP; also, we have limited the study of constraint solvers to complete approaches, based on the "branch and propagate" schema, and to constraints on finite domains.

To illustrate constraint programming, we have chosen a logic programming language. Indeed, it would have been interesting to illustrate constraint program-

ming with different languages, based on different paradigms. However, considering the small number of sessions, we have limited the study to one language. We have more particularly chosen Gnu-Prolog [8], a language developped by Daniel Diaz. Indeed, Gnu-Prolog is used to illustrate the Prolog course within the same course unit; it integrates a constraint solver over finite domains and provides a large number of built-in predicates for defining and solving constraints; finally, it is free and easy to install on most computers and operating systems.

3 Content of the sessions

The course on constraint programming is divided into 7 learning sessions of one hour.

Session 1 introduces constraint satisfaction problems.

In a first part, we introduce terminology and definitions: we define what is a constraint, and give an overview of the different kinds of constraints; we formally define what is a constraint satisfaction problem (CSP); we introduce the notion of variable assignment (complete or partial, consistent or inconsistent) and define what is a solution of a CSP; we introduce the concept of over and under constrained problems, and briefly discuss the main extensions to the CSP framework.

In a second part, we illustrate how to model a problem as a CSP through two examples. The first example is the well known n-queens problem: this problem is very simple to describe and allows us to introduce the fact that there may exist different CSP modelings for a same problem. The second example is the stable marriage problem [5], which has more practical applications.

Session 2 is a training session, where the student has to model 5 problems within the CSP framework:

- The first problem involves computing the set of coins that must be returned back by a slot machine, given a price and a quantity of inserted coins. This problem is modeled with integer variables and linear integer constraints. We then ask to add an optimization criterion in order to minimize the number of returned coins.
- The second problem is the classical map coloring problem.
- The third problem is a logical puzzle that has been proposed by Lewis Caroll in [3]: 6 friends have to decide what condiment to take (i.e., either

salt, or mustard, or both salt and mustard, or nothing) while satisfying 5 given logical rules. We ask for two different modelings for this problem: one which associates one 4-valued variable with each friend, and another one which associates one boolean decision variable for each condiment/friend pair.

- The fourth problem is the well-known "SEND + MORE = MONEY" cryptarithmetic puzzle, for which we ask for the two classical modelings: one with a single constraint that expresses the global sum constraint, and another one with carry variables and 5 sum constraints.

- The fifth problem is the well-known "zebra" puzzle, which involves associating a nationality, a colour, an animal, a favorite drinking, and a favorite cigarette tobacco to five consecutive houses, given a set of clues.

For each of these problems, students may ask for some help by clicking on a link that gives indications to help him identifying the variables, their domains and constraints holding between them.

Session 3 introduces basic principles of constraint solvers. We first describe the "generate-and-test" algorithm, which exhaustively enumerates all complete assignments until a solution is found, and we introduce the concept of search space of a CSP. We then describe the "simple-backtrack" algorithm, and we show that the integration of constraint checks within enumeration actually reduces the number of generated combinations. We then introduce the basic principle of constraint propagation and filtering algorithms, and the associated local consistencies, and we show how to integrate these filtering technics within the simple-backtrack algorithm. Finally, we discuss ordering heuristics, and we show that they may speed-up the solution process.

Each algorithm is first informally described. It is then given in imperative pseudo-code, and its run-time behavior is illustrated on the 4-queens problem.

Session 4 is a training session, where the student has to implement in Prolog the different algorithms introduced in session 3. The goal is to let the students have a better understanding and a first practice of the basic principles of enumeration and propagation. Another goal is to let them go deeper into the practice of the Prolog language, which has been studied previously during the five Prolog sessions, and to show that Prolog is very well suited to implement these algorithms.

To simplify constraints representation and consistency checking, we limit ourselves to binary constraints. The different algorithms that are implemented during the session are used to solve the n-queens problem, the map coloring problem, and the stable marriage problem (Prolog predicates describing these three problems are given to the student). Finally, the efficiency of the different algorithms is experimentally compared on the n-queens problem.

As this training session is not directly supervised by a teacher, we have to guide students progression. Hence, for each algorithm, we progressively introduce the different predicates to implement, and for each predicate to implement, we give its template, a description by means of the relationship between its arguments, and some examples of calls and answers.

Session 5 is dedicated to learning and using a constraint programming language, i.e., Gnu-Prolog.

In a first part, we introduce the built-in Gnu-Prolog predicates for defining finite domain variables, and constraints over finite domain variables, and for solving these constraints. For the main built-in predicates, we give a full description, and we illustrate them on different examples. We widely refer to the Gnu-Prolog on-line users' manual [8] for more information on other predicates.

In a second part of this session, we illustrate the built-in constraint predicates of Gnu-Prolog through the two examples introduced in session 1, i.e., the n-queen problem and the stable marriage problem.

Sessions 6 and 7 are training sessions, where students have to use Gnu-Prolog for solving the different problems introduced during session 2. For each exercise, we first recall the CSP modeling that has been designed during session 2. Then, we give some indications and we guide the students' progression for solving the CSP with Gnu-Prolog. In particular, we describe the main predicates that should be written, and for each of them we give its template and some examples of calls and answers.

4 Conclusion

First results. A first group of thirteen students has used this course this year. Some feedback on it has been provided through the questions they asked to the tutor... and more over through the questions they did not asked (!): they actually asked very few questions, on minor points only, and globally felt satisfied. For this first experiment, solutions to exercises were available on-line, and the tutor has not checked the solutions found by the students. As a consequence, we had

nearly no feedback about the difficulties they may have uncountered. For the next year, we have decided to deliberately hide the solutions to the students, and to ask them to send their own solutions to their tutor before sending them an "official" solution. This should allow us to evaluate more precisely the difficulties they encountered.

The whole "artificial intelligence" course unit has been ratified by a test. The questions about the constraint programming part mainly dealed with modeling a problem into a CSP. Answers were rather satisfactory, and the average score for the part concerning the constraint programming course has been slightly greater than the average score for the whole artificial intelligence course.

Related work and references. Anybody looking for information on constraint programming via Internet very quickly finds the "Online guide on constraint programming" written by Roman Bartàk [2], which is a very complete tutorial, and which has been a valuable source of inspiration for this course.

However, our goal —and therefore the resulting course— is rather different: we do not aim at training experts and at being complete on the subject, but we aim at training students to model constraint satisfaction problems, and to use a constraint programming language to solve them, within a limited amount of time (7 hours). Hence, our course only focuses on basic aspects and is not as complete as Roman Bartàk's guide on some points. In particular, we do not develop heuristic algorithms (such as local search) and approaches for solving over-constrained problems. As a counterpart, our course includes many exercices, and (try to) guide students to build their own solutions to these exercices. Also, our course includes some other points that are not developped in Roman Bartàk's guide: it introduces a constraint programming language, and illustrates how to use it to solve constraint satisfaction problems.

To write this course, we also took inspiration from many other tutorials and books on constraint programming, e.g., [1, 6, 7, 9, 10]. Students that have been appealed by this introductory course are refered to these books to actually become experts!

Finally, note that if this course has been designed for e-miage students, it is available (in french) at
`http://www710.univ-lyon1.fr/~csolnon/Site-PPC/e-miage-ppc-som.htm`

References

[1] K.R. Apt: Principles of Constraint Programming, Cambridge University Press, August 2003, 407 pages. ISBN: 0521825830.

[2] Roman Bartàk: On-line guide to constraint programming http://kti.ms.mff.cuni.cz/∼bartak/constraints/

[3] Lewis Carrol: Symbolic Logic, 1896 http://durendal.org/lcsl/

[4] e-miage: http://www.u-picardie.fr/∼cochard/IEM/

[5] Ian P. Gent and Patrick Prosser: an empirical study of the stable marriage problem with ties and incomplete lists, in the proceedings of ECAI 2002, IOS Press, pp 141–145

[6] François Fages: Programmation Logique par Contraintes, Collection "Cours de l'Ecole Polytechnique", Ellipses, Paris, 1996.

[7] T. Frühwirth and S. Abdennadher: Essentials of Constraint Programming, Springer Verlag, March 2003.

[8] GNU-Prolog: http://gnu-prolog.inria.fr/

[9] K. Marriott and P.J. Stuckey: Programming with Constraints: An Introduction, The MIT Press, 1998

[10] E. Tsang: Foundations of Constraint Satisfaction, Academic Press, 1993

Partial Specifications of Program Properties

Christoph Beierle, Marija Kulaš, Manfred Widera
Wissensbasierte Systeme, Fachbereich Informatik
FernUniversität in Hagen, 58084 Hagen, Germany
{beierle | marija.kulas | manfred.widera}@fernuni-hagen.de

Abstract

For the automatic revision of homework assignments in Prolog programming courses, in general one has to rely on testing or on validating programs with respect to a specification. Here, we present a pragmatic and flexible method for the partial specification of program properties. Within the AT(P) system, partial specifications can be used for automatic analysis of student solutions to Prolog exercises, yielding automatically generated feedback to the student. AT(P) is integrated into the Virtual University system of the FernUniversität in Hagen.

1 Introduction

For the automatic revision of homework assignments in Prolog programming courses, in general one has to rely on testing or on validating programs with respect to a specification. Here, we present a pragmatic and flexible method for the partial specification of program properties. Partial specifications can be used for automatic analysis of student solutions to Prolog exercises, yielding automatically generated feedback to the student.

This approach has been realized in the AT(P) system which is a Prolog instance of our more general AT(x) framework. Within the distance teaching environment where AT(P) is being used for teaching Prolog courses, three major requirements emerged:

- *Correctness*: when stating an error in a student's program, there indeed exists one.
- *Robustness*: guaranteed termination even for students' programs that cause runtime errors or infinite loops.
- *Understandable output*: messages of the system must especially aim at students.

In this paper, we provide an introduction to the AT(P) system. AT(P) was also presented at the 18. Workshop on Logic Programming in Potsdam [1]. Several parts of this paper describing the general AT(x) framework are taken from [1], while here backtracking tests are described in more detail, and new examples for partial specifications are given.

After giving an overview on the integration of AT(P) into the Virtual University system of the FernUniversität in Hagen, the general requirements of the central analysis component are described. The analysis of Prolog programs is based on the partial specification of program properties (procedural as well as declarative ones). In particular, we will give various examples of our approach to partial program specifications that can be expressed in AT(P), illustrating for instance how the messages provided by the system can be easily adapted to the students' needs.

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 [2, 12]. WebAssign 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 [6].

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 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.

AT(x) is divided into several components: the main work is done by the analysis component of the respective AT(x) instances (cf. Figure 1). 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. [13]) is therefore implemented in the target language (i.e. the language the programs to be tested are written in).

While for a description of the various other AT(x) components we refer to [1], the general requirements for the analysis components are summarized in the following section, whereas the Prolog analysis system for AT(P) is described in Sec. 4.

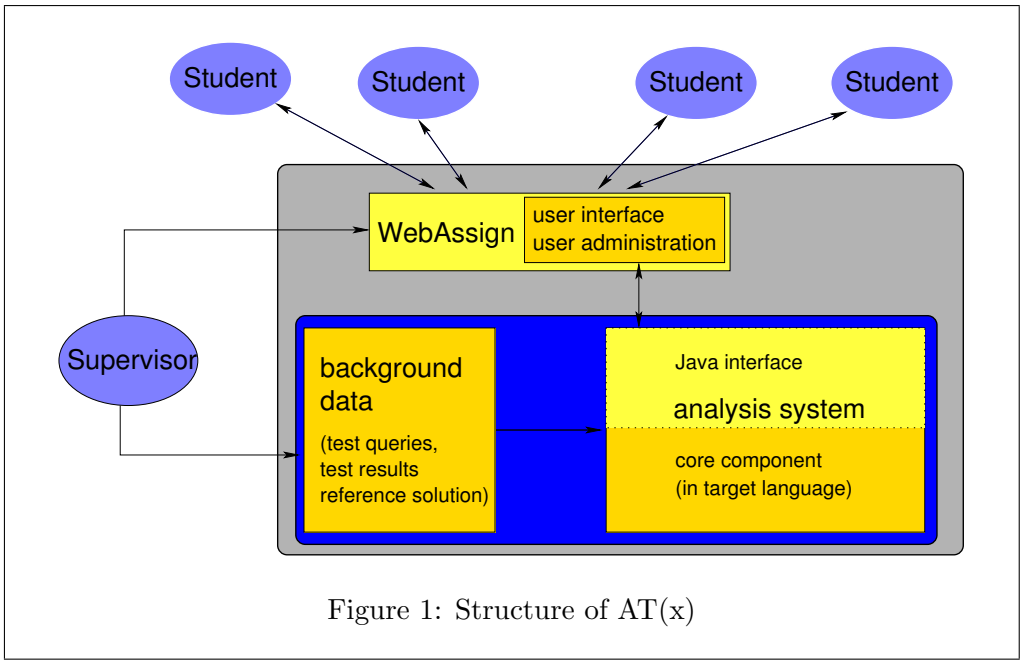

Figure 1: Structure of AT(x)

3 General 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 ter-

minating 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, an analysis component will have to rely on a safe approximation to the detection of non-termination (see Sec. 4), and the report to the student must clearly state the restricted confidence on any detected non-termination.

4 Analysis of Prolog Programs

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 essentially based on the TSP approach of H. Neumann [7] 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.

4.1 Annotation Tests

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 task of implementing a predicate *between/3* described as follows:

> Let N and M be integers 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 an integer between N and M.

A possible error of a student's program is to allow for too large values X in a call $between(X, N, M)$. We can detect and explain that by an annotation test where

- the test query is set to "*between(30, 10, 20)*",
- the success flag is set to expected failure,
- the error explanation text is "If X is greater than the uppe r bound M, *between(X,N,M)* must not succeed".

For instance, if this test fails, the system will generate the following output: *The following query succeded, though it should fail: between(30,10,20). Therefore, your program violates the following property:* If X is greater than the upper bound M, *between(X,N,M)* must not succeed. □

Apart from positive/negative annotations, the TSP system introduced *completeness annotations*, expressing constraints on the number of solutions to a query (see Sec. 5), and *file annotations*, enabling some handling of files. Further, *mode and example specifications* are special kinds of annotations as well.

4.2 Mode Tests

Mode tests form the class of tests that is 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:

   (a) Evaluate the query (as a backtracking test, to be explained below) 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 results of evaluation.

The generation of test queries uses a list of instantiated terms (*example 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:

```
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.[1]

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

[1] In contrast to the usual mode declaration $append(?Prefix, ?Suffix, ?Combined)$ found e.g. in [10], 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:

$$\begin{array}{ll} append(L1, L2, [a, b]), & append(L1, [a, b], [a, b]), \\ append([], L2, [a, b]), & append([], [a, b], [a, b])\,. \end{array}$$

The other example terms and mode declarations are processed analogously. □

4.3 Backtracking tests

Mode and completeness annotations give rise to so-called *backtracking tests*. The current test query (built from a mode annotation as shown above, or given in a completeness annotation) is evaluated within the original backtracking analyzer of [7]. The backtracking analyzer 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. The evaluation process is controlled by two limits. The backtracking limit L stems from the backtracking test in question. It is set to be $L := L_0 + 3$, where L_0 is the limit of the completeness annotation, or, in case of a mode test, $L_0 := 1$. So for a mode or completeness annotation, several extra solutions to each test query shall be sought, thereby giving more possibility to catch wrong solutions, or to detect universal termination. The second limit is the termination limit Max, the empirical estimation of the number of resolution steps and runtime calls needed. If L_0 is given, the execution of the test query ends at the latest after $L+1$ solutions have been found. Either way, the excution of the test query ends at the latest after $Max + 1$ steps.

A *mode error* is reported, if for some mode test query no student solution (i. e., no solution with respect to the student's program) is found, or some student solution is refuted by the reference program, or the termination limit is reached. Analogously for a *completeness error*.

4.4 Wrong and Missing Solutions

To determine wrong solutions, we have to evaluate each test query with respect to the student program, and then check the resulting substitutions with the reference program. The substitutions refuted by the reference program are wrong. If we also want to determine missing solutions, we need to evaluate each test query for the second time, now with respect to the reference program. Both programs, the reference program and the student's program, are held in memory in parallel using different modules.

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 $\Sigma = \{@, \#, \$\}$.

4.5 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.

4.6 Supervising Termination and Runtime Errors

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.

5 Partial Specifications of Program Properties

We will present concrete examples of specifications of program properties as they can be expressed in our system. They are provided by the tutor and present a powerful and very flexible means of expressing partial program specifications. Regarding the significance of "correct" specifications, one might ask what happens if a partial specification given by the tutor contains incomplete or wrong declarations of e.g. modes or examples. Of course, this could have dramatic effects leading to incorrect feedback to the students. However, this situation touches an inherent teaching problem: The teacher's instructions should be flawless. While it seems generally accepted that fully automatic generation of test cases is not feasible, AT(P) improves the homework assignment process compared to only manual correction and textual handout of reference solutions.

Factorial Suppose in a Prolog course, the predicate *fac/2* for computing the factorial of a natural number has been introduced, using the calling pattern *fac(+N,?F)*, i.e., if N and F are such that $F = N!$ then *fac(N,F)* holds provided

that the first argument is instantiated. Let us further assume that the following homework assigment is given to the students: *Define a predicate inv_fac/2 that computes both the factorial as well as its inverse, where at least one argument must be instantiated.*

Figure 2 gives a partial specification of *inv_fac/2*. In line 2, *modes/1* has a list of two mode terms expressing the intended usage of *inv_fac/2*. Together with the example term given in line 3, the test queries *inv_fac(5,F)*, *inv_fac(5,120)*, and *inv_fac(N,120)* are automatically generated.

```
% Import:  fac/2                                                    % 0
:- load_files([library('fac.pl')], [compilation_mode(assert_all)]).  % 1
modes([inv_fac(+N, ?F), inv_fac(?N, +F)]).                          % 2
examples([inv_fac(5,120)]).                                         % 3
testcase(complete(inv_fac(_N,1), 2, =)).                            % 4
testcase(pos_ann((inv_fac(0, F), F = 1),                            % 5
                 'inv_fac(0,F) => F = 1')).                         % 6
testcase(pos_ann((fac(6, F), inv_fac(X, F), X = 6),                 % 7
                 'F > 1 and fac(N,F) and inv_fac(X,F) => N = X')).  % 8
testcase(pos_ann((inv_fac(N, 720), fac(N, X), X = 720),             % 9
                 'inv_fac(N,F) and fac(N,X) => F = X')).            %10
testcase(pos_ann((inv_fac(5,F), fac(5,F)),                          %11
                 'inv_fac(N,F) => fac(N,F)')).                      %12
testcase(neg_ann(inv_fac(_N,100),                                   %13
                 'not fac(N,F) => not inv_fac(N,F)')).              %14
testcase(neg_ann(inv_fac(_N,0), 'not inv_fac(_N,0)')).              %15
testcase(neg_ann(inv_fac(_N,-1), 'F < 0 => not inv_fac(_N,F)')).    %16
testcase(neg_ann(inv_fac(-1,_F), 'N < 0 => not inv_fac(N,_F)')).    %17
```

Figure 2: Specification of program properties for *inv_fac/2*

Line 4 contains a so-called completeness annotation (cf. Sec. 4). The general form of a completeness annotation is

testcase(complete(+TestQuery, +Limit, +Op)).

which causes a check whether *TestQuery* produces a number N of solutions such that *N Op Limit* holds, with $Op \in \{<, =<, =, >=, >\}$. Thus, line 4 specifies that *inv_fac(_N,1)* has exactly 2 solutions (the factorial of both 0 and 1 yields 1).

Lines 5–17 contain four positive (5–12) and four negative (13–17) annotations. In general, for a positive annotation

testcase(pos_ann(+TestQuery, +Annotation)).

TestQuery should succeed. If it fails, *Annotation* is violated. For instance, if the query *inv_fac(N, 720), fac(N, X), X = 720* fails, then the annotation *'inv_fac(N,F) and fac(N,X) => F = X'* has been violated (lines 9-10). Note that the annotations are given as text strings, intended as error explanations for the students.

Analogously, the test query in a negative annotation should fail. For instance, if the query *inv_fac(_N,100)* succeeds, the annotation *'not fac(N,F) => not inv_fac(N,F)'* has been violated (lines 13-14).

Let us now suppose that the tutor wants the students to develop a version of *inv_fac/2* where both of its arguments may be uninstantiated. The specification of program properties given in Fig. 2 can be adapted to this modified task easily by just replacing line 2 by

```
modes([inv_fac(?N, ?F)]).
```

specifying the generalized mode situation.

Between A partial specification of program properties of the predicate *between/3* (Sec. 4) is given in Figure 3. Note that here, the tutor has choosen rather verbal annotations as error explanations, instead of more formal ones as for *inv_fac/2*. An example session involving *between/3* is given in the appendix.

Subterm Let the following homework assignment be given: *Define a predicate subterm/2 with calling pattern subterm(+SubTerm,+Term) that holds if SubTerm is a subterm of Term, where every term is a subterm of itself, and a subterm of an argument of some term T is a subterm of T as well. For instance, p(X) is a subterm of p(X) and of q(a,p(X),Y), but not a subterm of p(Y).* Figure 4 specifies various properties of *subterm/2*. If we want to extend the task by requiring that in the case of multiple occurrences of *SubTerm* in *Term* the predicate should succeed only once, we could adapt the partial program specification of Figure 4 by adding the completeness annotation

```
testcase(complete(subterm(a, p(a,g(a,a),a)), 1, =)).
```

Palindrom Let *alphabet([a, d, m]).* specify the alphabet of characters *a*, *d*, *m*. A *palindrom* is a sequence of characters (from this alphabet) that reads backwards the same ("Madam I'm Adam"). Figure 5 contains a partial specification of

```
modes([between(?_X, +_N, +_M)]).                                    % 1
examples([between(1,0,3)]).                                         % 2
testcase(complete(between(_X,100,102), 3, =)).                      % 3
testcase(complete(between(_X,5,5), 1, =)).                          % 4
testcase(neg_ann(between(_X,6,5), 'If the lower bound N is greater  % 5
 than the upper bound M, between(X,N,M) can not succeed')).         % 6
testcase(pos_ann(between(10,10,20), 'The lower bound N is           % 7
 between N and M')).                                                % 8
testcase(pos_ann(between(20,10,20), 'The upper bound M is           % 9
 between N and M')).                                               %10
testcase(pos_ann(between(20,20,20), 'N is between N and N')).      %11
testcase(neg_ann(between(20,30,40), 'If X is less than the lower    %12
 bound N, between(X,N,M) must not succeed')).                      %13
testcase(neg_ann(between(30,10,20), 'If X is greater than the upper %14
 bound M, between(X,N,M) must not succeed')).                      %15
```

Figure 3: Specification of program properties for *between/3*

palindrom/1 that holds if its argument is a palindrom, e.g. *palindrom([m, a, d, a, m])*. Note the use of the completeness annotation in line 3, requiring that the test query *?- palindrom(_L).* generates at least 10 solutions.

Cycles Assume that an undirected graph is given by stating its edges using *edge/2*, e.g.:

```
edge(a,b).     %
edge(a,c).     %          a ------ b
edge(c,d).     %          |        | \
edge(d,h).     %          |        |  \
edge(b,d).     %          c ------ d --h
edge(b,h).     %
```

Define a predicate *cycle/1* that holds if its argument is a *simple cycle*, i.e. a path in the graph with identical initial and final node where only one node occurs more than once in the path. For instance:

```
?- cycle([d, h, b, d]).
yes
```

```
modes([subterm(+_SubTerm, +_Term)]).                                    % 1
examples([subterm(p(X), q(a,p(X),y,p(p(X))))]).                         % 2
testcase(pos_ann(subterm(p(X), p(X)),                                   % 3
 'forall T: [atomic(T) or var(T) or compound(T)] => subterm(T,T)')).    % 4
testcase(neg_ann(subterm(p(X), p(Y)),                                   % 5
 'var(X) and var(Y) and not X == Y => not subterm(p(X),p(Y))')).        % 6
testcase(neg_ann(subterm(p(x), p(X)),                                   % 7
 'atomic(A) and var(X) => not subterm(p(A),p(X))')).                    % 8
testcase(neg_ann(subterm(p(x), p(q(x))),                                % 9
 'T1 does not occur as substring in T2 => not subterm(T1,T2)')).        %10
```

Figure 4: Specification of program properties for *subterm/2*

```
modes([palindrom(?_L)]).                                                %  1
examples([palindrom([m,a,d,a,m])]).                                     %  2
testcase(complete(palindrom(_L), 10, >=)).                              %  3
testcase(pos_ann(palindrom([]), 'palindrom([])')).                      %  4
testcase(pos_ann(palindrom([a]),                                        %  5
  'member(X,_Alphabet) => palindrom([X])')).                            %  6
testcase(pos_ann(palindrom([a, a]),                                     %  7
  'member(X,_Alphabet) => palindrom([X,X])')).                          %  8
testcase(pos_ann(palindrom([a, a, a]),                                  %  9
  'member(X,_Alphabet) => palindrom([X,X,X])')).                        % 10
testcase(pos_ann(palindrom([d, m, a, a, m, d]),                         % 11
  '[forall X: member(X,A) => member(X,_Alphabet)] and reverse(A,B)       % 12
   and append(A,B,C) => palindrom(C)')).                                % 13
testcase(neg_ann(palindrom([m, a, b, a, m]),                            % 14
  'not_member(X,_Alphabet) and member(X,L) => not palindrom(L)')).      % 15
testcase(neg_ann(palindrom([m, a, d, a, m, a]),                         % 16
  'L = [A|_R] and length(L,N) and even(N) and last(L,B)                 % 17
   and not A == B => not palindrom(L)')).                               % 18
testcase(neg_ann(palindrom([m, a, d, a, d]),                            % 19
  'L = [A|_R] and length(L,N) and odd(N) and last(L,B)                  % 20
   and not A == B => not palindrom(L)')).                               % 21
```

Figure 5: Specification of program properties for *palindrom/1*

```
modes([cycle(?_cycle)]).                                              % 1
examples([cycle([h,d,c,a,b,h])]).                                     % 2
testcase(complete(cycle([h|Rest]), 6, =)).                            % 3
testcase(neg_ann(cycle([]), 'The empty path is not a cycle')).        % 4
testcase(neg_ann(cycle([a,a]),                                        % 5
  'not edge(A,A) => not cycle([A,A])')).                              % 6
testcase(neg_ann(cycle([a,b,a,c,a]), 'Z = [A|Rest] and Rest           % 7
  contains A more than once => not cycle(Z)')).                       % 8
testcase(neg_ann(cycle([a,b,d,h,b,a]), 'Z = [A|Rest] and Rest         % 9
  contains a node other than A more than once => not cycle(Z)')).     %10
testcase(neg_ann(cycle([a,b,d,c]), 'Z = [A|Rest] and last(Rest,B)     %11
  and not A == B => not cycle(Z)')).                                  %12
testcase(neg_ann(cycle([a,h,a]),                                      %13
  'not A == B and not edge(A,B) => not cycle([A,B,A])')).             %14
```

Figure 6: Specification of program properties for *cycle/1*

```
?- cycle([a, b|Rest]).
Rest = [a]     ;
Rest = [d, c, a]     ;
Rest = [h, d, c, a]
```

Figure 6 contains a partial specification of *cycle/1*. For instance, line (2) specifies that there are exactly 6 simple cycles with start node *h*. Please note that the testcases in Figure 6 are rather fine-tuned towards the given graph; however, given the flexibility of our specification approach, it is easy to use also more general ones.

6 Conclusions and Further Work

The AT(P) system is fully implemented and operational. The analysis component runs under SICStus Prolog based on the SUN Solaris or Suse Linux operating system and, via its Java interface component, serves as a client for WebAssign. AT(P) is being used in the framework of a course on deduction and inference systems at the FernUniversität Hagen, as well as in a course on logic and functional programming. After the usage of AT(P), feedback from the students was very positive in general; a more detailed evaluation of the system has still to be done.

Of the several annotation types shown in Sec. 5, the modes and examples have proven to be especially useful. Compared to the little effort needed to formulate modes and examples, there is ample reward in automatically detected errors. Also, along with the introduction of the system TSP, at the heart of AT(P), our exercises have become more precise, thus reducing the risk of misunderstanding with the students. Some reference solutions benefitted as well: by testing against alternative versions, several improvements could be achieved. Finally, the modes and examples are useful documentation of the predicates in question.

In the area of testing and analysis of Prolog programs there have been many proposals, ranging from theorem proving (eg. [11]) to various forms of debugging (eg. [3]) and systematic testing. The proposals differ along several axes: static or run-time analysis, restricting the target language or not, expressiveness of the annotation language. For a comparison of our AT(P) approach to various systems like LPTP [11], GUPU [8], ADVICE [9], CIAO-Prolog [4], NOPE [5], or the TSP system [7], we refer to [1].

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 [7]. TSP offers a variety of different tests and turned out to be extremely stable; the partial specifications presented here were adapted from [7].

References

[1] C. Beierle, M. Kulaš, and M. Widera. A pragmatic approach to pre-testing Prolog programs. In D. Seipel, M. Hanus, U. Geske, and O. Breitenstein, editors, *Proc. 15th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2004) and 18th Workschop on Logic Programming (WLP 2004)*, pages 102–114. Technical Report 327, Univ. Würzburg, March 2004.

[2] J. Brunsmann, A. Homrighausen, H.-W. Six, and J. Voss. Assignments in a Virtual University – The WebAssign-System. In *Proc. 19th World Con-*

ference on Open Learning and Distance Education, Vienna, Austria, June 1999.

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

[11] 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.

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

[13] Manfred Widera. Testing Scheme programming assignments automatically. In S. Gilmore, editor, *Trends in Functional Programming*, volume 4. Intellect, 2004. (to appear).

Appendix A

Let us assume that the following program is submitted for the *between/3* (see Example 1):

```
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 AT(P)'s output is the following:

```
The following query failed, though it should succeed:
   between(10,10,20)
Therefore, your program violates the following property:
   The lower bound N is between N and M
-------------------------------------------------
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)
```

AT(P) 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.

Teaching Prolog Programming at the Eötvös Loránd University, Budapest

Tibor Ásványi
Faculty of Informatics, Eötvös Loránd Univ.
Budapest, Pázmány Péter sétány 1/c, H-1117
asvanyi@inf.elte.hu

Abstract

At the Eötvös Loránd University (Budapest) we have two courses on Prolog programming, especially for program designer (MsC) students.

Our main objective is to help the students find the subproblems in their projects to be solved with Prolog, and enable them to write the necessary code. At first our main concern is to give clear notions. Next we focus on the technical details of writing small programs. Then we incorporate the questions raised by developing big applications.

In our first course we give an overview of the history of logic programming (LP). We introduce the basic notions of this area, and we discuss the widely used LP language, *Prolog*, emphasizing a kind of pragmatic programming methodology which helps us to develop correct and effective programs. We talk many short examples over, in order to make the students feel the taste of LP in small.

In our second course we focus on writing bigger programs and using advanced Prolog programming techniques like *generate and test* techniques, *exception handling*, writing and using *second-order predicates*, splitting our programs into *modules*, using *partial data structures*, *d-lists*, and *logic grammars*. We discuss many typical problems to be solved with logic programs.

1 Introduction

Logic programming is taught at our university from the early eighties.

In the beginning the legendary MProlog system of Péter Szeredi and his colleagues was used.

Later some of the author's colleagues changed to Turbo Prolog, because of its effectivity, good development environment and graphics. It is still used in the teacher-training courses, although it is considered obsolete now.

The program designers' Prolog courses are in the competence of the author, who changed to SICStus Prolog in 1994, because it seemed to be the best implementation with affordable costs, available for many platforms.

However, during the semesters these questions are not in the centre. We focus on the following aspects:

1. We present many problems which can be solved elegantly and effectively with Prolog.
2. We show how to split the program into modules, taking advantage of existing code, especially of libraries.
3. We discuss how to refine the predicates while maintaining the finiteness of the search tree, controlling the data flow, and organizing the deterministic and nondeterministic parts.

On the language issues, first we concentrate on the standard features of SICStus, but later we take into consideration the implementation specific details, like its module system and libraries.

The Prolog courses are based on semesters about *algorithms and data structures*, about *first order logic and resolution*, and about *artificial intelligence*. It is also supposed that the students have practice in developing structured, procedure based programs, they have written some programs consisting of many components, they are familiar with some assembly language, and they have basic knowledge on formal languages and compilers.

During the Prolog courses we use *SICStus* with *Emacs* interface. In such a way we have a complete development evironment. First we constrain ourselves to using the *ISO standard* [1] subset of this implementation, in order to help the students write portable programs. But later we take into consideration the implementation specific details, that is, the *module system*, the *libraries*, the *DCG rules*, and the *hook predicates* of SICStus, in order to help the students write large applications.

There are yet other two courses on logic programming at our university. The first one is about the theoretical foundations of LP, the second is a comparative study of LP languages.

2 Motivation: Why Prolog?

Given a set of logic statements defining a model, we can formulate queries on (unknown) objects satisfying known relations. A constructive answering process can be considered a computation: This is a fundemantal idea of logic programming.

Therefore a logic program is similar to a theorem proving system. It is a set of axioms together with a control component. But its control component is much simpler. The computation process can be followed and guided by the programmer, its termination can be guaranteed, and its costs are predictable.

Prolog is a simple and old, but successful LP language. It offers the necessary simplifications and additions compared to a pure LP system, so that the programmer have an effective and flexible tool.

3 Beginner's Course

We roughly follow the first two parts of *The Art of Prolog* [4], but we give less theory, because we have a special theoretical course on this topic. Other main sources are [1, 3, 5, 2].

Our own textbook is [6]. We concentrate on a practical Prolog programming methodology, in order to help the students develop their own applications.

1. First – with a historical background – we introduce the notions of the *logic* and *control* components of the logic programs. The logic component is detailed first, because of the students' theoretical background: It is shown how a relation is refined.

2. The run of the program is a constructive proof of the query, and query-driven (top-down) proof, that is, goal-reduction is to be preferred in general: It corresponds to the refining of relations and it is usually more effective than other strategies. In order to illustrate the run of the program we introduce the *search-trees* and mention that there are the same solutions in the different search-trees.

3. We introduce recursive programs and compound terms, especially lists. We emphasize that usually the *search tree* of the queries must be *finite*, and give examples how to prove this.

4. Next we go on to pure Prolog, define the Prolog machine, detail the *occurs check problem* (explain why to omit this check), discuss how to turn STO (subject to occurs check) programs into NSTO (not STO) programs, so that the possible occurs checks are restricted to the calls to the predefined predicate `unify_with_occurs_check/2` [1].

5. Then we discuss two usual optimizations of the Prolog machine: *first argument indexing* and *last call optimization*. We write some simple programs taking advantage of these. We emphasize that these optimizations are not parts of the ISO standard.

6. Next we introduce *disjunctions, conditionals* (`(If->Then;Else)` and `(If->Then)`), *negation*, green and red *cuts*, and discuss the safe use of cuts, comparing it with conditionals, and used together with indexing. We argue that the run of a deterministic predicate should never leave choice point.

7. Then we discuss the *meta-logical* (arithmetic, type-checking, term comparing, term manipulating) predicates (with many examples), and the meta-variable facility, especially with `findall/3`. We show that considering the data flow of the program becomes especially important, if we use meta-logical predicates.

8. We finish the beginner's course with the *extra-logical* predicates. First we consider the I/O predicates needed in real applications. Second we distinguish the *static* and the *dynamic* predicates together with the standard built-ins accessing and manipulating our programs. We emphasize that the use of the modifications of the programs should be restricted to generating *lemmas* and negative lemmas, and passing information among the branches of the search-tree.

4 Advanced Course

We roughly follow the third and fourth parts of *The Art of Prolog* [4]. Other main sources are [1, 3, 6, 5, 2]. Still we do not have our own textbook.

1. At the advanced course we start with some classical problems to be solved with *nondeterministic programming*: the eight queens problem, map colouring, and some logic puzzles. These problems are in the heart of LP, therefore this chapter seems to be a good start at the advanced course.

   Here we introduce *exception handling* in order to protect our programs against incorrect input.

2. Then we consider the *second-order predicates* of Prolog, especially those collecting the solutions of a goal. These are useful in collecting the solutions of a nondeterministic search, and will be needed in the advanced search tecniques to be introduced soon.

3. At this point our programs are going to become complex enough to be divided into components. Therefore, next we discuss the different *module systems* used in the Prolog implementations, and the problems with the second-order predicates, while using the module system of SICStus [3].

4. Then we use *partial data structures* to define dictionaries, *d-lists*, and especially *queues*. Here we reconsider the problem of writing optimal Prolog code in general, and especially in SICStus.

   The use of *partial data structures* is not easy for the students, therefore it is delayed until this point. However it is necessary for the subsequent themes, which are illustrations of some widely used algorithms from the fields of *artificial intelligence*, and of *compiler writing*.

5. Next we go on to different state-space problems. We introduce a set of graph-searching techniques, especially backtracking, depth-first and breadth-first search (with stacks and queues). We apply *algoritm A* and *algoritm A** to solve some problems like the classical *fifteen puzzle*, because these algorithms provide safe heuristic search methods.

   At this stage we split the programs into modules, and we define the graph-searching modules independently from the actual problem.

6. Then we discuss some two-player games and apply the classical *alpha-beta pruning*.

7. Last we introduce *logic grammars*, especially *DCG rules*. We discuss the whole compiler (consisting of six modules) of a simplified Pascal-like language. The parser is defined with a DCG grammar generating the syntax tree of the program (except if there are syntax errors: then these are handled). The code generator is another DCG grammar. It takes the syntax tree and generates an abstract assembly code, and a dictionary. Its result is taken by the assembler (a third DCG grammar) which generates the object code.

It may be unusual that we emphasize the module system of SICStus, although there is no module system in the ISO standard [1]. But modularization and writing generic purpose modules is a general trend in software development. For example, let us see, how to define and use a generic graph-search module. In order to concentrate on the organization of the program we can choose the simplest one, the backtracking graph-search strategy.

```
:- module( bts, [ init_graph_search/1, graph_search/1 ] ).

:- use_module( library(lists), [ member/2, reverse/2 ] ).

init_graph_search( File ) :-
    use_module( File, [ arc/2, start/1, goal/1 ] ).
```

```
graph_search(Path) :- start(A), gs(A,[],Path).

gs(A,Ancestors,Path) :- goal(A), reverse([A|Ancestors],Path).
gs(A,Ancestors,Path) :-
    arc(A,B),
    B \= A, \+ member(B,Ancestors),    % no loop
    gs(B,[A|Ancestors],Path).
```

Module `bts` is a generic one. This means that it does not know the graph it searches on. Somebody must call `init_graph_search(File)`. Its run will import the necessary predicates (`arc/2,start/1,goal/1`) into module `bts`.

Module `graph` contains the necessary predicates, that is, the description of the graph. Again, it is independent from the graph-searching method used. Somebody must call `init_graph(File)`. Its run will import predicate `graph_search/1` into module `graph`.

```
:- module( graph, [ init_graph/1, arc/2, start/1, goal/1,
                                          graph_search/3 ] ).

init_graph( File ) :-
    use_module( File, [ graph_search/1 ] ).

:- dynamic(start/1).    start(a).
:- dynamic(goal/1).     goal(e).

graph_search(Start,Goal,Path) :-
    retractall(start(_)), retractall(goal(_)),
    asserta(start(Start)), asserta(goal(Goal)),
    graph_search(Path).

arc(a,b).    arc(b,c).    arc(b,d).
arc(c,a).    arc(c,e).    arc(d,e).
```

```
%              ----------
%             /          \
%            V            \
%            a----->b----->c
%                   |      |
%                   |      |
%                   V      V
%                   d----->e
```

The main module just loads the components of the program, and makes them known to each other.

```
:- module( main, [ graph_search/1, graph_search/3 ] ).

:- use_module( bts, [ init_graph_search/1, graph_search/1 ] ).
:- use_module( graph, [ init_graph/1, graph_search/3 ] ).

:- init_graph_search(graph), init_graph(bts).
```

Having loaded the main module, the program can be used as follows.

```
| ?- graph_search(Path).
Path = [a,b,c,e] ? ;
Path = [a,b,d,e] ? ;
no
| ?- graph_search(c,e,Path).
Path = [c,a,b,d,e] ? ;
Path = [c,e] ? ;
no
```

5 Final Remarks

We have two semesters on Prolog programming for program designer students at our university. These cover the standard material of such courses with some modification: There is an emphasis on how to develop effective, big programs with Prolog.

Our semesters are special lectures. The students can choose them as parts of their MsC program. Both of the courses take 20 hours of teaching, and are worth 2 ECTS credits. (There are 30 credits per semester in our system.) In a year, approximately 80 students finishes the first, and 30 students finishes the second

course. In a year, 80-100 students receives MsC. This means that most of them have some knowledge on LP. About the same number of students stop at BsC. Unfortunately they do not hear about LP, and we do not see how to change this situation.

We do not teach writing expert systems at the Prolog courses, because there is a special course on expert systems, and (among other shells) we teach the Prolog extension *flex* there. The theoretical foundations and the comparison of LP languages are covered by other two semesters. Still we would like to start a new course on constraint programming, with high emphasis on CLP. Another possibility is to include Péter Szeredi's special courses on LP (from the Technical University of Budapest) in our selection.

References

[1] Deransart P., Ed-Dbali A.A., Cervoni L., *Prolog: The Standard (Reference Manual)*, Springer-Verlag, 1996.

[2] O'Keefe R. A., *The Craft of Prolog*, The MIT Press, Cambridge, Massachusetts, 1990.

[3] *SICStus Prolog 3.11 User's Manual*, Swedish Institute of Computer Science, PO Box 1263, S-164 28 Kista, Sweden, 2003.
(`http://www.sics.se/isl/sicstuswww/site/documentation.html`)

[4] Sterling L., Shapiro E., *The Art of Prolog* (Second Edition), The MIT Press, London, England, 1994.

[5] Szeredi P., Benkő T., *Deklaratív programozás: Bevezetés a logikai programozásba (Declarative Programming: Introduction into Logic Programming)*, BME, Budapest, 2000.

[6] Ásványi T., *Logikai programozás (Logic Programming)*, **in:** "Programozási nyelvek (Programming Languages)", (**editor:** Nyékyné Dr. Gaizler Judit), pp. 637-684. Kiskapu Kft. Budapest, 2003.

Prolog as Description and Implementation Language in Computer Science Teaching

Henning Christiansen
Roskilde University, Computer Science Dept.,
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

Abstract

Prolog is a powerful pedagogical instrument for theoretical elements of computer science when used as combined description language and experimentation tool. A teaching methodology based on this principle has been developed and successfully applied in a context with a heterogeneous student population with uneven mathematical backgrounds. Definitional interpreters, compilers, and other models of computation are defined in a systematic way as Prolog programs, and as a result, formal descriptions become running prototypes that can be tested and modified by the students. These programs can be extended in straightforward ways into tools such as analyzers, tracers and debuggers. Experience shows a high learning curve, especially when the principles are complemented with a learning-by-doing approach having the students to develop such descriptions themselves from an informal introduction.

1 Introduction

Teaching of theoretical aspects of computer science to university students that do not necessarily possess a solid mathematical background may sound like a contradiction. The Advanced Studies in Computer Science at Roskilde University, Denmark, is a part of long tradition of interdisciplinary studies in which the same courses often are offered for classes of students with different backgrounds such as Natural Science, Humanities, or Social Sciences. Certain issues that are important for all sorts of teaching become extra critical in this context, and furthermore stressed by the fact that a tradition of 50% student project work throughout the studies leaves only very little time for regular courses. First of all, the presentation needs to be appealing and fruitful for every single student in this heterogeneous audience. Secondly, extreme care must be made in the selection of topics in order to provide a coherent course with a reasonable covering,

considering that each course has few nominal hours. Finally, each course must be designed as a component of a full education comparable with any other five-year university education with computer science as a major subject.

This paper gives an overview of a teaching methodology developed under these conditions in which Prolog plays the combined role of as a study object and, more importantly, as a meta-language for describing and experimenting with different models of computation, including programming language semantics and Turing machines, and tools such as tracers and debuggers. The approach has been developed and successfully applied during the 1990s and used in courses until recently; a full account of the approach can be found in a journal paper [2] that also gives a more comprehensive set of references to related approaches; a locally printed textbook in Danish is available [1].

In the following, we analyze the qualities of Prolog that we have relied on in this approach, and we show how definitional interpreters, compilers and other models of computation can be defined in a systematic way as Prolog programs based on a general model of abstract machines. In this way, formal descriptions become running prototypes that are fairly easy to understand and appealing for the students to test and modify. The approach has turned out to be highly effective when combined with learning-by-doing which has been applied for type-checking and implementation of recursive procedures. A brief listing is given of other items treated in a course based on the these principles, and a sample course schedule is shown.

2 Qualities of Prolog in relation to teaching

Prolog is a wonderful programming language for any teacher of computer science: Students with or without previous programming experience can learn to write interesting programs with only a few hours of introduction and guided experiments in front of a computer. A substantial subset of Prolog exposes a mathematically and intuitively simple semantics and makes a good point to emphasize the distinction between declarative and procedural semantics, and thus also to isolate various pragmatic extensions from the core language.

Computer science as university subject contains many aspects where Prolog can be interesting, independently of whether the students intend to use Prolog in their future careers. First of all, Prolog is an obvious second programming language that shows the diversity of the field for student brought up with a language such as Java. Prolog is a type-less language in which any data structure has a denotation and with no need for constructors and selection methods as these are embedded in Prolog's unification. Java, on the other hand, requires the

programmer to produce large collections of classes, interfaces, methods, and a test main method before anything can be executed. The conflict between flexibility, conciseness, and semantic clarity on the one hand, and security and robustness on the other is so obviously exposed in this comparison. Prolog's application as a database language is well-known and we shall not go into details here; in section 5 we mention briefly how an introduction to databases has been incorporated in our approach.

A study of Prolog motivates also considerations about the notion of a meta-language: `assert` and `retract` take arguments that represent program text, the same goes for Prolog's approximation to negation-as-failure which essentially is a meta-linguistic device within the language. The problematic semantics of these features gives rise to a discussion of what requirements should be made to a meta-linguistic representation. Operator definitions in Prolog comprise syntactic meta-language within the language, and are also a perfect point of departure for a detailed treatment of priority and associativity in programming language syntax.

In general, we have relied on the following detailed properties of Prolog.

- Prolog terms with operator definitions provide an immediate representation of abstract syntax trees in a textually pleasing form; see the following expression which with an operator definition for ":=" is a Prolog term:
  `a:= 221; b:= 493; while(a =\= b, if(a>b, a:= a-b, b:= b-a))`

- Structurally inductive definitions are expressed straightforwardly in Prolog by means of rules and unification, e.g.,
  `stmnt(while(C,S),···):- condition(C,···), stmnt(S,···), ···.`

- Data types for, say, symbol tables and variable bindings, are easily implemented by Prolog structures and a few auxiliary predicates.

- Specifications are directly executable and can be monitored in detail using a tracer; they can be developed and tested incrementally and interactively. Students can easily modify or extend examples and test their solutions.

  Prolog invites to an interactive and incremental style of program development, not only for students but also for the teacher to do this during the lecture using a computer attached to a projector.

- The characterization of various pragmatic issues can be developed in direct relation to "ideal" formal descriptions. An interpreter, for example, is easily extended into a tracer or debugger, and code optimization can be incorporated in a small compiler written in Prolog.

- Last but no least: Prolog appears as an easily accessible framework compared with, say, set and domain theory. Although basically representing the same universal concepts, the combined logical and operational nature of Prolog-based specifications gives an incomparable intuitive support.

3 A basic model of abstract machines

An unsophisticated model of abstract machines is a central element in our methodology, used for the general characterization of computer languages and computational models.

A particular *abstract machine* is characterized by its *input language* which is a collection of phrases or sentences, a *memory* which at any given time contains a value from some domain of values, and finally a *semantic function* mapping a phrase of the input language and memory state into a new memory state. For simplicity, output is not explicit part of the definition but considered as part of the "transparent" memory whenever needed.

The framework includes a general notion of *implementation* of one machine in terms of another, and three different modes are defined, *interpretation*, *translation* and use of *abstraction mechanisms* in standard programming languages. Interpreters and translators themselves, as well as program modules, can be explained as particular abstract machines.

Abstract and concrete syntax are introduced and distinguished in an informal way, and the representation of abstract syntax trees by Prolog terms (as above) is emphasized. The abstract syntax of a context-free language is characterized by a recursive Prolog program consisting of rules of the form

cat_0`(`op`(T`$_1$`,...,T`$_n$`)):-` cat_1`(T`$_1$`),...,`cat_n`(T`$_n$`).`

where *op* names an operator combining phrases of syntactic categories cat_1, ..., cat_n into a phrase of category cat_0.

Syntax-directed definitions can be specified by adding more arguments corresponding to the synthesized as well as inherited attributes of an attribute grammar [5]. Consistent with our abstract machine model, we introduce what we call a *defining interpreter* which to each syntax tree associates its *semantic relation* of tuples $\langle s_1, \ldots, s_k \rangle$ by predicates of the form

cat_i`(`*syntax-tree*`,`s_1`,...,`s_k`)`

As an example, a defining interpreter for an imperative language may associate with each statement a relation between variable state before and after execution, which for a statement such as "`x:= x+1`" contains among others the following tuples: ⟨`[x=7],[x=8]`⟩, ⟨`[x=1,y=32]],[x=2,y=32]`⟩,

4 Imperative and procedural languages

In the following we show how standard programming languages are characterized in our Prolog-based style, indicating the spirit in which it is communicated in the teaching. We proceed by introducing a defining interpreter for a simple machine-like language giving a continuation-style semantics for jumps and control points. This serves the dual purposes of making the semantics of such languages explicit and of introducing continuations as programming technique and semantic principle. Next is shown a defining interpreter for while-programs and a compiler of while-programs into machine language. Finally we describe an assignment where the students developed type checker and interpreter for a simple Pascal-like language from a brief, informal introduction.

4.1 A defining interpreter for a machine language

The following Prolog list is an abstract syntax tree for a program in a simplified machine language. Presenting this sample to the students is sufficient to indicate the existence of an abstract machine, and it gives good sense to execute this program by hand on the blackboard from the intuition provided by the instruction names.

```
[     push(2),
      store(t),
   7, fetch(x),
      ...
      equal,
      n_jump(7)]
```

The semantics of such programs assumes a stack (that we can represent as a Prolog list) and a storage of variable bindings (represented conveniently as lists of "equations", e.g., `[a=17,x=1,y=32]`). The central predicate in a defining interpreter is the following. The first argument represents a sequence of instructions (a continuation) to be executed and the second one passes the entire program around to all instructions to give the contextual meaning of labels.

`sequence(`*Seq*, *Prog*, $Stack_{current}$, $Store_{current}$, $Stack_{final}$, $Store_{final}$`)`

The meaning of simple statements that transform the state is given by tail-recursive rules such as the following: Do whatever state transition is indicated by the first instruction and give the resulting state to the continuation. Example:

```
sequence([add|Cont], Prog, [X,Y|S0], L0, S1, L1):-
     YplusX is Y + X,
     sequence(Cont, Prog, [YplusX|S0], L0, S1, L1).
```

The unconditional jump instruction is defined as follows; it is assumed that the diverse usages of the `append` predicate have been exercised thoroughly with the students at an earlier stage.

```
sequence([jump(E)|_], P, S0, L0, S1, L1):-
     append(_, [E|Cont], P),
     sequence(Cont, P, S0, L0, S1, L1).
```

Executing a few examples, perhaps complemented by a drawing on the blackboard — and within a few minutes the students have grasped the principle of a continuation and continuation semantics.

The remaining rules that complete the interpreter are straightforward.

A little aside can be made, turning the interpreter into a functioning tracer by adding the following rule as the first one to the interpreter:

```
sequence([Inst|_],_,_,_,_,_):- write(Inst), write(' '), fail.
```

Students are given the following exercises that serve the twofold purpose of familiarizing them with the material and introducing other important aspects: extend language and interpreter with instructions for subroutines; write a Prolog program checking that labels are used in a consistent way; write a Prolog predicate that optimizes selected subsequences of instructions; design and implement an extension of the tracer with debugging commands.

4.2 A defining interpreter for while-programs

As a next step up the ladder of languages moving away from the machine and closer to "problem-oriented" languages, we consider while-programs whose semantics also can be specified in terms of a defining interpreter. A defining interpreter consists of the following predicates.

`program(`*program*`,` *final-storage*`)`
`statement(`*statement*`,` *storage-before*`,` *storage-after*`)`
`expression(`*expression*`,` *storage*`,` *integer*`)`
`condition(`*condition*`,` *storage*`,` {`true`,` `false`}`)`

Most rules are straightforward, the most complicated one being the following defining the meaning of a while statement.

```
statement( while(Cond, Stm), L1, L2):-
  condition(Cond, L1, Value),
  (Value = true -> statement((Stm ; while(Cond, Stm)),L1,L2)
                ; L1=L2).
```

The following exercises are given to the students: run a sample program including a while loop with Prolog's debugger switched on and record all primitive actions; extend the language with expressions of the form `result_is(` *statement*`,` *variable*`)`; extend the language with a `for` loop; extend the interpreter with a simple tracing facility.

4.3 A compiler for while-programs

The structure of our defining interpreters can also be adapted to describe compilers. Above, we considered a semantics for while-programs defined in terms of state transformations and now we consider an alternate semantics capturing meanings by means of sequences of machine instructions.

Two auxiliary predicates are introduces, one for creating unused machine language labels and another one to facilitate the composition of sequences of instructions; illustrated below. The following rule specifies the compilation of a while statement.

```
statement( while(Cond, Stm), C):-
      condition(Cond, CondC),
      statement( Stm, C1),
      new_label(Lstart), new_label(Lend),
      C <-  Lstart + CondC +
                    n_jump(Lend) +
                    C1 +
                    jump(Lstart) +
            Lend.
```

The compiled code for the while statement is composed by the code for its constituents, two new labels created by `new_label` and specific instructions; the predicate denoted by "`<-`" puts together the sequence indicated by "`+`" in its second argument and unifies it with the first argument. Notice that `n_jump` is a conditional jump to the specified label whenever the previous computation has placed a value representing false on top of the stack. The code produced can be executed by the interpreter shown in section 4.1. As before, exercises are given that involve testing and extending this compiler in various ways.

4.4 A learning-by-doing approach to recursive procedures and type-checking

The detailed semantics and implementation of recursive procedures and type-checking are usually consider very difficult by students. We have had good success with these topics by means of a larger learning-by-doing assignment continuing the material presented so far.

The students were presented for a simple Pascal-like language by means of example programs with a recursive quicksort program as a prototypical representative. Type requirements and a standard stack-based implementation principle for recursive procedures were described informally, and the assignment was to implement both type-checker and compiler in Prolog.

The prescribed time for the work was one week on half time, including writing a small report documenting the solutions. The most experienced students had type checker and interpreter running after four or five hours, and all students in a class of some 30 students solved the task within the prescribed time. All solutions were acceptable and there was no obvious difference between those produced by students with a mathematical background and by those without.

5 Other course elements

Here we list other topics integrated with the previous material in different versions of our course; more details including program samples can be found in [2].

Logic circuits modeled in Prolog is a standard example used in many Prolog text books. This is obvious to apply in our context due to the meta-linguistic character (modeling the language of logic circuits).

LISP modeled with assert-retract. Function definitions and variable bindings are implemented using Prolog's assert-retract. Illustrates dynamic binding and different levels of binding times plus introduces functional programming. The use of assert-retract as opposed to explicit state arguments makes it possible to model an interactive Lisp environment with few lines of codes.

Turing machines. An introduction to computability theory is given, based on Turing machines and Turing completeness. An interpreter made up by a few lines of Prolog is an excellent way to illustrate a Turing-machine and to provide a truly dynamic model, especially when a tracing facility is added. The existence of the interpreter shows that Prolog is Turing-complete, and having played with it makes it easier for the students to understand the proof of undecidability of the halting problem.

Vanilla and Prolog source-to-source compilation. The familiar Vanilla self-interpreter for Prolog [7] is a perfect example to illustrate the notion of a self-interpreter.

Appearing a bit absurd and useless to the students in the first place, they begin to see the point of a self-interpreter when a few lines of additional code makes it into a tracer and debugger. Source-to-source compilation is illustrated in terms of a profiling tool that inserts additional code to record the number of entrances, successes and failures of each clause in a Prolog program.

Relational algebra in Prolog. The course described here has in some years been integrated with a standard database course. As an introduction to relational database technology, students were given the assignment of implementing an interpreter for relational algebra. The conditions were the same as for the task on type-checking and recursive procedures described above, one week on half time, including writing a small report documenting the solutions. This task has been given to several classes of students and all students usually succeed in producing an acceptable solution, although `join` often causes problems.

Syntax analysis. Traditional methods for lexical analysis and parsing are integral components of our course. Prolog is used as a ready-at-hand tool for the students to implement finite state machines, deterministic as well as nondeterministic. Top-down parsing is illustrated perfectly by Prolog's built-in Definite Clause Grammars [6], and bottom up-parsers by an analogous grammar formalism CHRG [3] developed on top of Constraint Handling Rules [4] which is a recent extension to some Prolog versions that provides a natural paradigm for bottom-up evaluation. Now quick and effective introductions can be given to standard implementation principles for finite state machines and parsing.

Dissecting a Prolog implementation in Java. As a conclusion of the course, the students are shown a full implementation in Java of a subset of Prolog, including lexical analysis, parsing, representation of abstract syntax trees in an object-oriented language, and an interpreter which exposes a detailed implementation of Prolog's unification procedure.

6 A sample course schedule

The following table shows the schedule for a version of a course designed according to our methodology as it was given in spring 2001. The actual course has changed slightly from semester to semester so not all items mentioned above are included. The course corresponds to 25% of a student's work in one semester (7.5 ECTS) and is concentrated on 10 full course days. Each course day consists of lectures

and practical problem solving related to the day's lecture. A considerable amount of homework is expected from the students.

1	Introduction: Abstract and concrete syntax, semantics, pragmatics, language and meta-language. Prolog workshop I: The core language, incl. structures.
2	Prolog workshop II: Lists, operators, assert/retract, cut, negation-as-failure.
3	Abstract machines: Definitions of a.m., interpreter, translator, etc. Prolog workshop II contd.
4	Language and meta-language, Prolog as meta-language. Semantics of sequential and imperative languages; defining interpreters and a small compiler.
5	Declarations, types, type checking, context-dependencies, recursive procedures.
6	Introduction to and practical work with large exercise: do-it-your-self recursive procedures, interpreter and type checker.
7	Conclusion and comments to large exercise. Turing-machines, decidability and computability, Turing universality, the halting problem, Turing machines in Prolog.
8	Constraint logic programming: Introduction to CLP(R) and CHR; CHR Grammars for bottom-up parsing.
9	Syntax analysis: Lexical analysis and parsing; recursive-descent parsing
10	Overview of phases in a traditional compiler. Dissection of an implementation of Prolog in Java. Evaluation of the course.

7 Conclusion

We have explained a methodology based on a combination of a simple, underlying model of abstract machines and the use of Prolog as general definition and implementation language. Prolog is well suited for this purpose: Conceptual simplicity and high expressibility with a core language consistent with a subset of first-order logic; syntactic extensibility that allows a direct notation for abstract syntax trees in a textually acceptable form; a rule-based structure that fits perfectly with an inductive style of definition. Last but not least: Prolog is an interactive language that appeals to incremental development, testing, and experimentation with an extremely short turn-around time from idea → implementation → observation → revision or extension of idea.

Our experience have shown that theoretical issues of computer science can be taught in this way in an entertaining and concrete way which, unlike traditional

approaches, appeals to a wide range of students for which a uniform mathematical background cannot be taken for granted.

A critical remark may be that this form of learning is very compact, with many important aspects covered by one minimalist and seemingly innocent example as was the case with the interpreter for machine language. One might fear that students tend to remember only the example and not the points that the teacher had in mind. We have not applied any scientifically based evaluation principle, but it is our clear impression that the practical work in exercises and larger assignments serves fully to avoid this potential danger. Informal evaluations with the students have indicated a high degree of satisfaction with the teaching principle. Especially the larger learning-by-doing assignments (type-checking plus recursion; relational algebra in Prolog) were characterized as difficult and challenging, but also some of the most interesting ones from which the students had learned quite a lot.

Acknowledgment: This research is supported in part by the IT-University of Copenhagen.

References

[1] Henning Christiansen. Sprog og abstrakte maskiner, 3. rev. udgave [in Danish; eqv. "Languages and abstract machines"]. Datalogiske noter 18, Roskilde University, Roskilde, Denmark, 2000.

[2] Henning Christiansen. Teaching computer languages and elementary theory for mixed audiences at university level. *Computer Science Education Journal*, 14, 2004. To appear.

[3] Henning Christiansen. CHR Grammars. *Int'l Journal on Theory and Practice of Logic Programming*, 2005. To appear.

[4] Thom Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[5] Donald Knuth. Semantics for Context-Free Languages. *Mathematical Systems Theory*, 2:127–145, 1968.

[6] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis — a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[7] D. H. D. Warren. Implementing Prolog - Compiling Predicate Logic Programs – Volumes 1 & 2. D.A.I. Research Report 39, 40, University of Edinburgh, May 1977.

Teaching Logic Programming at the Budapest University of Technology

Péter Szeredi

szeredi@cs.bme.hu

Department of Computer Science and Information Theory,
Budapest University of Technology and Economics
H-1117 Budapest, Magyar tudósok körútja 2.

Abstract

The paper describes courses related to Logic Programming at the Budapest University of Technology and Economics. We present the layout and the contents of the main such course, entitled Declarative Programming, as well as the tools used in teaching this subject. We then give a brief outline of the elective courses and other educational activities in the subject area.

1 Introduction

Logic Programming and the Prolog language has been taught at several Hungarian universities since the mid 1970's. I have been involved in teaching LP from the very beginning, first at the Eötvös Loránd University, Budapest, and later at the Budapest University of Technology and Economics (BUTE).

The paper gives an overview of courses related to Logic Programming at the Faculty of Electrical Engineering and Informatics at BUTE. These courses are primarily intended for undergraduate students of informatics, although students of mathematics and electrical engineering sometimes also take these courses.[1]

In Section 2 the declarative programming course is discussed in detail. Section 3 describes the elective courses related to LP. Note that details of one of these courses, on constraint programming, have recently been published in [11]. Section 4 discusses educational activities other than courses, such as student projects, student conference papers, as well as theses related to LP. The paper is concluded with a brief summary discussion.

[1] Note that at present the undergraduate studies at BUTE span 5 years and lead to an MSc degree.

2 The Declarative Programming course

Declarative programming is part of the compulsory studies for students of informatics at BUTE. The course is scheduled in the fourth semester, i.e. the spring semester of the second year, although the students are allowed to take the course earlier or later. It is preceded by programming language courses on C, C++, and Java, as well as by a course on mathematical logic, which introduces, among others, the theory of logic programming.

The Declarative Programming (DP) course covers the two main declarative paradigms. Péter Hanák teaches functional programming, using the MOSML dialect of the SML language, while Péter Szeredi is responsible for the logic programming part, which focuses on Prolog and uses the SICStus Prolog implementation [9].

The course in this setup has been taught in every semester since 1994. It evolved from a wider subject entitled "Programming Paradigms", which involved imperative, object-oriented and functional programming styles.[2]

The semester normally consists of 14 weeks. The DP course has two lectures of 2*45 minutes each week. The structure of the course is the following:

Lecture 1:	Introducing the declarative programming paradigm
Lectures 2-8:	Logic Programming, part 1
Lectures 9-15:	Functional Programming, part 1
Lectures 16-21:	Logic Programming, part 2
Lectures 22-27:	Functional Programming, part 2
Lecture 28:	Summary, outlook

Very unfortunately, the DP course has no laboratory exercises. In order to encourage students to do some programming tasks we hand out during the course several minor assignments as well as a single major assignment (see 2.2). We have also developed a computer tool, called ETS (Electronic Teaching aSsistant, or Elektronikus TanárSegéd in Hungarian) [3], to support Web-based student exercising, assignment submission/evaluation, marking, etc.

Although there is a clear division of the LP and FP parts of the course, these build on each other. We reuse concepts and techniques introduced in the other part, e.g. those of tail recursion, accumulators, construction and decomposition of data structures using pattern matching (unification). The two parts are also linked by the common major assignment, and occasionally by common minor assignments and exam tasks, see 2.2.

[2] Actually, the course run under the name "Programming Paradigms" until autumn 1999.

We will now focus on the common and logic programming parts of the DP course.

2.1 The topics covered by the course

This section briefly describes the topics covered by the course, lecture by lecture, following the schedule of the 2004 spring semester.

Lecture 1. Introducing the declarative programming paradigm. A very simple declarative subset of the C language.[3]

Lecture 2. Introductory Prolog examples (family relations, summing numbers in binary trees), Prolog as a subset of logic, declarative semantics.

Lecture 3. Procedural semantics of Prolog, execution models (goal-reduction and procedure-box models).

Lecture 4. Data structures, unification, the logic variable.

Lecture 5. Operators, disjunction, negation, if-then-else.

Lecture 6. Lists, basic list handling library predicates.

Lecture 7. Example: finding paths in a graph, using various representations.

Lecture 8. Prolog syntax summary.

Lecture 16. Pruning the search space. Control predicates.

Lecture 17. Determinism, indexing, tail-recursion, accumulators.

Lecture 18. Rewriting imperative programs to Prolog, collecting and enumerating solutions.

Lecture 19. Meta-logical built-in predicates.

Lecture 20. Modularity, meta-predicates, meta-programming, dynamic predicates.

Lecture 21. Definite clause grammars, "traditional" built-in predicates.

Lecture 28. Brief outlook on LP extensions (external interfaces, coroutining, constraints).

The material covered in the lectures is made available to the students in the form of a textbook manuscript [12], updated every 2–3 years.

[3]The subset supports only integer types, function declarations and calls, `if` and `return` statements, the `+`, `-`, `*`, `/`, and `%` arithmetic operators, and the six comparison operators.

2.2 Programming examples and assignments

This section describes some of the programming examples and assignments used in the course.

Examples. At the very first lecture we show a fairly limited natural language conversation system (in Hungarian), which can remember statements and reply to questions related to these. This example program is then discussed in detail towards the end of the course (lecture 21, DCGs).

In lecture 4 we illustrate the symbolic processing capabilities of Prolog via a simple example program for building arithmetic expressions using the four main arithmetic operators (`+`, `-`, `*`, and `/`), each of which can be used zero or more times. The task is to build such an expression, so that both its value and the numbers it contains are given in advance (each given number should be used exactly once). A specific instance of this task, building an expression valued 24 from numbers 1, 3, 4, and 6, seems to be quite a difficult puzzle for humans ...

In lecture 19 we present a two-slide program implementing the standard ordering relation of Prolog (`@<`) (with the exception of variable ordering). This sample program re-iterates the definition of the standard ordering using a wide range of meta-logical predicates (`functor`, `arg`, `atom_codes`).

Minor Assignments. The minor assignments are non-compulsory, and have to be submitted via the ETS system. The solution of the student is then automatically tested on a set of predefined test cases and the results are sent back to the submitter via email.[4]

The minor assignments can be re-submitted any number of times, up to the deadline (which normally is about two weeks after the announcement). After the deadline all submissions are re-tested on a new test-set.

The first minor assignment in the 2004 spring semester was related to the declarative C language subset introduced in the first lecture. The students were asked to write a fairly complex function in this C subset, `palindrome(a)`, which returns the smallest integer `b` such that `a` written in base `b` is a palindrome.

The next two minor assignments were about binary trees storing key-value pairs, described by the following Mercury-style type declaration:

[4] As suggested by one of the reviewers, it would be better to show the results in the browser, rather than by sending them in an email. The reason for the latter choice is that some assignments (most notably the major ones) have longer running times (up to several minutes), so it would be infeasible to have the student wait for the results. On the other hand, the exercising facility of the ETS, see 2.3, is fully interactive, and provides immediate feedback for the student.

```
:- type tree(T) ---> node(tree(T),tree(T),T)
                   | empty.
```

In assignment 2 we ask for a predicate `tree_pair(Tree, Key, Value)` to be written. This predicate should enumerate all `Key`s and corresponding `Value`s stored in a given `Tree` of type `tree(pair(KType,VType))`.

In assignment 3 a predicate `pairs_keys_values(Tree, TreeK, TreeV)` has to be submitted. All three arguments are isomorphic trees, each node of `Tree` contains a `Key-Value` pair, where `Key` and `Value` are in the corresponding nodes of `TreeK` and `TreeV`. The predicate (of course) should be capable of both splitting a tree of pairs, and building such a tree from separate key- and value-trees.

The last and most complex minor assignment addresses issues related to meta-logical predicates and meta-programming. A predicate is to be written, which is capable of transforming an arbitrary Prolog term by wrapping all "good" lists `GL` occurring in it (at any depth) into a `good(GL)` structure. A list is good if it is ground and all its elements satisfy a condition. The condition is given as a term, which should be called using the `call/2` non-standard meta-predicate.

Major Assignment. There is a single, non-compulsory major assignment issued in a given semester, which can be be submitted in Prolog, or SML, or both languages. Solving the major assignment requires much more effort than the solution of the minor ones.

The major assignment is very often a logic puzzle. Several such assignments from earlier years have been published in the paper on the BUTE constraint course [11] (as the DP major assignment is normally re-used in the constraint course of the subsequent semester).

The major assignment issued in the 2004 spring semester was the Clouds puzzle:

> A rectangular board of size $n * m$ units is given. The task is to mark certain fields (unit squares) of the board as belonging to a cloud. The following conditions are known to always hold:
>
> 1. Clouds occupy an area of rectangular shape and their width and height is at least two units.
> 2. No clouds touch each other, not even diagonally.
>
> To solve the puzzle one is given the following pieces of information:
>
> 1. the size of the board;
> 2. the number of cloudy fields in certain rows/columns of the board;

3. the presence or absence of a cloud at certain fields of the board.

Figure 1 shows an example puzzle and its (unique) solution.[5] A number is given by each row and column. If this number is non-negative, it is equal to the count of the cloudy fields, otherwise the count is not known. Cloudy fields known in advance are represented by the '+' character, while fields known to be non-cloudy (clear) by '-'. In the solution (right hand side) '#' represent cloudy fields and '.' clear sky.

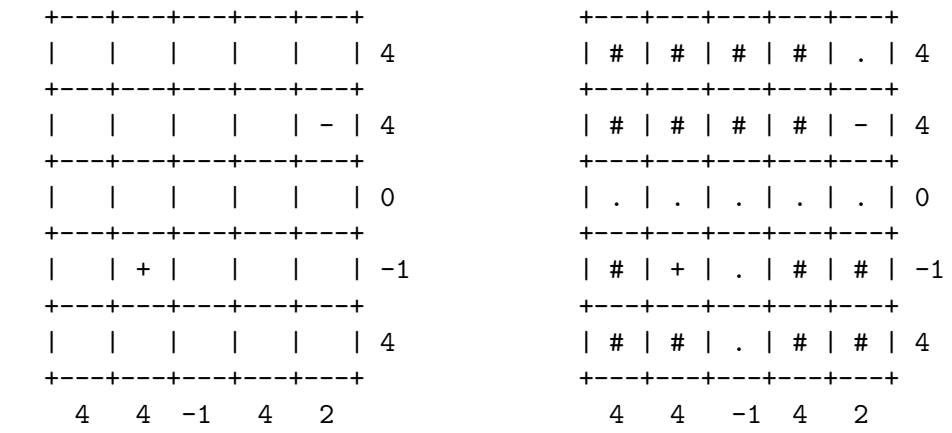

Figure 1: A Clouds puzzle and its solution

We use at least three distinct sets of test cases for assessing the students' programs. The first set is distributed with the announcement of the assignment, so that students can use it in the process of program development. The second set is used when the students hand in their solutions. The results of this test are sent back to the students via email. As for the minor assignments, students can hand in as many versions of their programs as they wish, but only the latest version will be used for the final test. This test is run on the third set of test cases, which has a similar difficulty level as the earlier ones. Students also have to submit a documentation via the ETS system.

The best solutions for the major assignment, which solve all the test cases of the final test, participate in the so-called "ladder contest". These programs are tested against bigger and more difficult test cases. The authors of the programs which achieve highest places in the ladder contest, get extra points.

In spring 2004 the largest test-case in the ladder test-set was of size 25*18, with 92 given fields. Only a single student submission was capable of solving this

[5]Naturally, a puzzle can have multiple solutions.

test case within the prescribed 120 second timeout.

2.3 Teaching tools

We now briefly introduce the tools and utilities used by the students and/or the lecturers during the DP course.

The ETS electronic teaching assistant. The web-based ETS system provides the following services:

- access to a database of students of the course, together with their results,
- assignment submission and automatic testing,
- facilities for student exercising.

The ETS system has been developed by the course lecturers together with several talented students of the course, who devoted their student research work [1] or Master's Thesis [2] to this subject.

We now focus on the student exercising facilities of ETS. The system supports various exercise schemes for both the Prolog and SML languages. As worked out in [2], a scheme describes how the exercise is presented to the student as well as how the student answer is processed. Correspondingly, a scheme is associated with a web-form, which includes the generic text of the exercise. This form is then specialised using a database of concrete exercise instances.

At present the following Prolog schemes are supported:

- Canonical form: the student is requested to type in the canonical form of a Prolog term (as if printed by `write_canonical`).
- Execution: The student should decide what will be the result of executing a given goal. There are three sub-schemes, depending on the determinism, number of variables, etc.:
  - Success/failure/error: The goal is deterministic and it can fail or raise an exception. In case of success, the value of a single variable is asked for.
  - Multiple variables: The goal is deterministic and it always succeeds. The student is asked to supply the substitutions of all variables. Typically used for unification exercises.

  - All solutions: The goal is possibly non-deterministic with a single variable. The student is asked to enumerate all solutions of the goal, in proper order.[6]

- Programming: The student has to write a Prolog program following a given specification.

Orthogonally to the schemes, the exercises are grouped into topics, which correspond to sections of the material taught. For example, the topic of lists may contain exercises in various schemes: canonical form (of lists), execution and programming (of list processing predicates).

For most of the topics the exercises come in several difficulty levels: easy, medium, and hard. Students can prescribe the difficulty level of exercises they want to practice.

We now give a few examples of various exercises in the ETS system.

- "Canonical form" scheme, lists and operators topic, hard exercises:

```
[[],[]]     - (1,2)     1 - - 1.
```

- "Success/failure/error" scheme, unification topic:

```
| ?- [X,1|X] = [_,_].
| ?- [X,a,X] = [1,2,a,1,2].
```

- "Multiple variables" scheme, unification topic:

```
| ?- g(1+2+3, [a,b]) = g(X+Y, [U|V]).
| ?- h([H, G], H*G) = h([Q/1|R], P/Q*3).
```

- "Success/failure/error" scheme, list processing and control predicates topics:

```
| ?- length(X, 1), member(a, X).
| ?- member(X, [1,2,3]), !, X < 3.
```

- "Multiple solutions" scheme, list processing topic:

[6] This sub-scheme could be eliminated in principle by encapsulating the goal into a `findall`. However this type of exercise is issued much earlier than `findall` is taught.

Program:	`app([X|L1], L2, [X|L3]) :-`
	`  app(L1, L2, L3).`
	`app([], L, L).`

Goal:	`| ?- app(L, [a|_], [a,b,a,b,a]).`
Task:	List the substitution(s) of `L` separated by `;`'s.

(The correct answer is: `[a,b,a,b];[a,b];[]`.)

Further details and examples regarding the capabilities of ETS can be found in [3].

Other tools. The declarative C language subset, used in the initial lecture has also been implemented and its interpreter has been made available to the students. This interactive implementation has been written in Prolog and it works by translating the C code to Prolog predicates.[7] Students thus can get acquainted with declarative programming and interactive function invocation in an already familiar environment of the C language.

Further tools serve for the visualisation of Prolog execution. First I developed a simple, non-graphical tool for drawing the search tree of a pure Prolog program. Subsequently, as a student research project, this was extended to general Prolog programs and a graphical interface was also provided [7].

Regarding tools used by the lecturers, we should mention that the slide presentations of the whole course (both Prolog and SML) have been developed using the `xdvipresent` tool [5]. Another, very important utility serves for detecting plagiarism in the student assignments. This tool has been developed in a student research project in 2000 [6] and since then it have excellently served the purpose of deterring students from copying each others work. The basic idea is to transform the programs into call-graphs and compare these graphs, rather than the program texts. Of course, this tool can not be used for simple assignments, where the code to be written consists of a few predicates or functions.

At present, the plagiarism detection tool contains front-ends for both SML and Prolog, but there are plans to extend it for further languages.

3 Elective courses

There are two elective courses closely related to logic programming:

[7] In the future we plan to hand out this compiler to students, and to use it in the second part of the course for illustrating the advantages of Logic Programming in compiler writing, cf. [13].

- "Highly efficient logic programming" held each year since 1997,
- "Selected topics from logic programming", student seminar, held in 2001 and 2003.

Both courses are single semester ones, with one lecture/seminar of 90 minutes per week.

The "Highly efficient logic programming" (HELP) course tries to show two facets of efficient logic programming:

- Making programs run faster by creating a streamlined and clean LP language, as exemplified by the Mercury project [10].
- Making programs run faster by adding more reasoning capabilities as shown by the constraint logic programming (CLP) approach.

However, the emphasis of the course shifted towards CLP, which takes about 11–12 lectures, while the basic outline of Mercury is presented in the remaining 2–3 lectures. The CLP part is based on SICStus Prolog, discussing all four constraint libraries present: CLP(R/Q), CLP(B), CLP(FD) and CHR.

There is an important link between the DP and the HELP course: the major assignment of the DP course is re-issued in the subsequent semester as the HELP assignment. It is not uncommon that the CLP program written as the HELP assignment is two orders of magnitude faster than the Prolog solution of the same student.

Further details on the HELP course can be found in [11].

In the seminar entitled "Selected topics from logic programming" students themselves present their account on a topic of their interest, be it a paper, some experience with an application or a programming language, etc. The lecture themes come from various areas of logic programming: theory, parallelism, object-oriented, graphical, and web-related extensions, abstract interpretation, tracing, as well as overviews of concrete Prolog implementations.

In the first few weeks I hold the seminars, so that the students have time for preparing their presentations. In both editions of the course so far, these lectures were about Prolog implementation techniques, primarily discussing the WAM approach. This is followed by student presentations of 45–90 minutes, depending on the topic. The students are asked to prepare their presentation at least one week in advance, and to discuss it with me; this ensures fairly good quality. The seminars are often followed by a lively discussion. Overall, the students were quite satisfied with the seminar.

In spring 2004 a new course on the Semantic Web was set up by myself and Gergely Lukácsy. Although this course is rather loosely related to LP, it is worth

mentioning that several students have prepared their assignment, a tableaux based reasoner for a Description Logic, in Prolog.

4 Other educational activities

There are several types of activities at BUTE, where enthusiastic students can explore some topics in greater depth.

Directed Projects. In semesters 8 and 9 all students of informatics have to do a Directed Project with a weekly load equivalent to six 45 minute lectures. I had the pleasure of leading about a dozen such student projects over the last seven years, in various areas of logic programming. Some of the more interesting ones are listed below:

- Applying CLP(FD) for scheduling plastic moulding machines — Tamás Benkő, 1997
- Interfacing Prolog to Corba — Gábor Gesztesi and Gábor Marosi, 1997–98
- Debugging CLP(FD) programs — Dávid Hanák and Tamás Szeredi, 2000 (this led to the development of a new SICStus library [4])
- Using CHR for reasoning in Description Logics — Bence Szász, 2002
- A Prolog based RDF reasoning system — Gergely Lukácsy, 2002
- A Prolog-Java interface using sockets — Péter Biener, 2003

Masters' Theses. It is quite often the case that students chose their work in the Directed Project as the basis for their MSc Thesis. In fact this happened to all but the Corba project in the above list. Some further Theses I supervised are listed below:

- Implementation of a constraint reasoning system — Tamás Rozmán, 1997
- Conversion of document description languages — Zsolt Lente, 2000
- Knowledge-based tools for information integration — Attila Fokt, 2000
- Computer Support for Declarative Programming Courses — Dávid Hanák, 2001

- Verification of object-oriented models using constraints — Péter Tarján, 2001
- Transforming object-constraints to logic — Károly Opor, 2002
- Logic-based methods for planning queries on heterogeneous data sources — Tamás Lukács Berki, 2003

Student Research Projects. There is a long tradition of Hungarian students doing a research project, in addition to their curricular activities. Such student research work is assessed at yearly Student Conferences, to which students have to submit a paper (usually 40–60 pages) and also deliver a 20 minute oral presentation.

Separate Student Conferences are held every autumn within each Faculty of BUTE. In November 2003, there were 118 presentations involving about 170 students in 11 sections, in the Student Conference of the Faculty of Electrical Engineering and Informatics. Best papers get first, second and third prizes, these rewards are taken into account e.g. in the admission procedure of PhD studies. Paper which have received a first prize participate in the biennial National Students Conference.

The following Student Conference papers were presented in the area of (constraint) logic programming:

- Solving a stock exchange allocation problem (using CLP), Dániel Varró, 1998, I. Prize.
- Conversion of SGML languages, Zsolt Lente, 1999, II. Prize.
- Comparison of source program structures, Gergely Lukácsy, 2000, I. Prize, Rector's Special Prize; I. Prize at the National Student Conference, 2001
- Efficient access of an object-oriented database from logic programs, Ambrus Wagner, 2000
- A Web-based student exercising system for teaching programming languages, András György Békés, Lukács Tamás Berki, 2001
- Intelligent querying and reasoning on the Web, Gergely Lukácsy, 2002, I. Prize; Special Prize of the Hungarian W3C Office at the National Student Conference, 2003
- Visualisation of Prolog program execution, Tamás Nepusz, 2003, II. Prize

- Using abstract interpretation in SICStus Prolog, Balázs Leitem, 2003, III. Prize

5 Summary

I think it is a very good thing that each and every student of informatics at BUTE (which currently means about 400 students every year) has to get acquainted with the basics of logic and functional programming. I only hope that this remains so when the current five year curriculum (leading to an MSc degree) gets divided into two parts, following the Bologna principles.

It looks that the introductory DP course can attract the attention of talented students, who then get involved in further elective courses. A possible explanation of this is that, as opposed to other programming courses, such as C, Java, etc., the DP course introduces a programming style previously unknown to most students, and at the same time it shows some problems where this new paradigm can be successfully used. The major assignment, which is not compulsory, but is practically required for achieving the highest grade, gets the attention of the best students as a task which looks very difficult to solve using traditional, imperative languages. The ladder contest for the major assignment adds to this the thrill of a peer-to-peer competition. Consequently, quite a few students make an attempt to solve the major assignment, and so get some practice in declarative programming.

The major assignment of the DP course has almost always been a constraint problem. I often mention to students at the DP exam that their solution can be made hundred times faster using constraint techniques, which they can get acquainted with in the HELP course. This "advertisement" seems to work: for example, there are now 59 students enrolled for the 2004 autumn HELP course (as a reference, 480 students were enrolled in the 2004 spring DP course).

Both the logic programming and the CLP courses focus more on the practical programming aspects, and less on the theoretical ones. Although I agree that it would be beneficial to make the courses a bit more balanced in this respect, I think that the primary interest of the students, at least here in Hungary, lies in the practical applicability of LP, rather than in its theory. It may be interesting to note, however, that in the last edition of the LP seminar there was a student-initiated lecture on theoretical foundations of (C)LP, based on the first chapters of [8].

The student evaluations[8] of the courses described in this paper have almost always been positive. Nevertheless there are quite a few issues that need im-

[8]Students at BUTE are asked to fill in a questionnaire for each lecturer of each course every semester.

provement.

Most importantly, the better integration of the functional and logic programming parts of the DP course is always on the agenda. Teaching a single language encapsulating both functional and logic programming aspects (such as Oz, or Mercury) would help to avoid the (mostly syntactic) confusion of the present Prolog + SML setup. On the other hand, the novel languages mentioned have much less industrial acceptance at present. Also, switching to a new language would require a major investment on the lecturers' part, which we cannot afford presently. Therefore, we now remain with the two languages, and try to link the two topics in the lectures, pointing out the similarities and differences of Prolog and SML.

There is also scope for improvement regarding the utilities and tools used in the courses. Students without Internet access would prefer a stand-alone exercising tool to the present Web-based one. The set of exercise schemes and exercises needs further expansion, especially regarding exercises involving program writing. The administrative part of ETS supports a single course in a single semester, it would be good to eliminate both limitations. Several improvements are underway regarding the tool for Prolog execution visualisation. The plagiarism detection tool has to be extended to support the declarative subset of the C language, introduced recently.

As the referees rightly pointed out, the paper would have benefited from the inclusion of more details on the mathematical logic course, as well as on the functional programming part of the DP course. Unfortunately, the strict time constraints did not allow for these to be included.

6 Conclusions

In the paper I tried to give an overview of the role logic programming plays in undergraduate education and research at the Budapest University of Technology and Economics. I hope that the experiences reported here can be of help in LP education at other universities.

Acknowledgement

I am indebted to Péter Hanák, who, exactly 10 years ago, invited me to teach declarative programming at BUTE, and with whom I could share the joy and troubles of this difficult task. I am also most grateful to all the enthusiastic student helpers (we had about 50 of these!), and especially to those who helped in the development of various utilities and tools: András György Békés, Tamás

Benkő, Lukács Tamás Berki, Dávid Hanák, Gergely Lukácsy, Tamás Nepusz, and Tamás Rozmán.

Thanks are also due to the anonymous referees for their helpful remarks.

References

[1] András György Békés and Lukács Tamás Berki. A Web-based student exercising system for teaching programming languages (in Hungarian), 2001. Students' Conference, BUTE, Budapest, Hungary.

[2] Dávid Hanák. Computer support for declarative programming courses (in Hungarian), 2001. MSc Thesis, BUTE, Budapest, Hungary.

[3] Dávid Hanák, Tamás Benkő, Péter Hanák, and Péter Szeredi. Computer aided exercising in Prolog and SML. In *Proceedings of the Workshop on Functional and Declarative Programming in Education, PLI 2002, Pittsburgh PA, USA*, October 2002.

[4] Dávid Hanák and Tamás Szeredi. FDBG, the CLP(FD) debugger library of SICStus Prolog. In Susana Muñoz Hernández and José Manuel Gómez-Pérez, editors, *Proceedings of the Fourteenth International Workshop on Logic Programming Environments (WLPE'04)*, Saint-Malo, France, September 2004.

[5] Manuel Hermenegildo. Slide presentations using latex/xdvi, 2003. CLIP Group, School of Computer Science, Technical University of Madrid, `http://clip.dia.fi.upm.es/Software/xdvipresent_html`.

[6] Gergely Lukácsy. Comparison of source program structures (in Hungarian), 2001. National Students' Conference, Eger, Hungary.

[7] Tamás Nepusz. Visualisation of Prolog program execution (in Hungarian), 2003. Students' Conference, BUTE, Budapest, Hungary.

[8] Ulf Nilsson and Jan Maluszynski. *Logic, Programming and Prolog (2nd ed).* John Wiley, 1995.

[9] SICS, Swedish Institute of Computer Science. *SICStus Prolog Manual, 3.11*, June 2004.

[10] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

[11] Péter Szeredi. Teaching constraints through logic puzzles. In Krzysztof R. Apt et al., editor, *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2003, Selected Papers*, volume 3010 of *Lecture Notes in Computer Science*, pages 196–222. Springer, 2004.

[12] Péter Szeredi and Tamás Benkő. *Introduction to logic programming (in Hungarian)*. Budapest University of Technology and Economics, Faculty of Electrical Engineering and Informatics, 1998, 2001, 2004. Study-aid for the Declarative Programming course. Manuscript.

[13] David H. D. Warren. Logic programming and compiler writing. *Software Practice and Experience*, 10:97–125, 1980.

A Logic Programming E-Learning Tool For Teaching Database Dependency Theory

Paul Douglas
University of Westminster, London, UK
P.Douglas@wmin.ac.uk

Steve Barker
King's College, London, UK
steve@dcs.kcl.ac.uk

Abstract

In this paper, we describe an "intelligent" tool for helping to teach the principles of database design. The software that we present uses PROLOG to implement a teaching tool with which students can explore the concepts of dependency theory, and the normalization process. Students are able to construct their own learning environment and can develop their understanding of the material at a pace that is controlled by the individual student.

1 Introduction

We describe a tool that we have developed and have used to help university-level students to learn certain essential notions in database schema design, specifically the *normalization* [1] process, which is based on the underlying concept of *dependency theory* [1]. We regard the learning tool as a piece of intelligent software, where the term "intelligent" is interpreted by us as the capability of responding to a student's input (in the form of a database design problem *chosen by the student*) with a solution based entirely on that input. There are no "standard" problems or solutions provided with the software, and the software is able to explain each step of the solution process if the student chooses to exploit this option. It follows that the software is capable of providing either a quick check of a student's own work, or a fuller teaching facility.

PROLOG is used for the implementation of the software to provide the capability of checking a student's work and either confirming that the work is correct, or indicating why it is not. PROLOG has been widely used for implementing items of educational software (see, for example, [2] and [3]) and is appropriate for the teaching tool that we have developed because it permits a rule-engine to be exploited to intelligently interpret and respond to student inputs. The database

design algorithms that we have chosen to implement also have a natural translation into PROLOG code.

A number of excellent textbooks on dependency theory already exist. Nevertheless, although textbooks can offer very good coverage of the material on dependency theory, they offer only limited forms of interactivity and limited scope for students to test their own understanding of database design principles. Many textbooks provide no practical exercises and, even when they do, these exercises are often limited in size and sophistication. Moreover, textbooks that do provide exercises do not necessarily provide "solutions", so students cannot determine whether they were able to "solve" the problems.

Some courses that teach relational databases take the approach that the use of a schema design tool will almost always deliver a schema that is in *third normal form (3NF)* [1], and that teaching dependency theory is not really necessary (see, for example, [4]). However, we disagree with this point of view. There are many aspects of relational database technology that are directly related to the data dependencies that exist within a database, and students cannot properly consider these issues without a proper understanding of dependency theory. The use of design tools also suggests that a relational database schema has been somehow finalized once it has been put into third normal form, leaving students with even less understanding of normal forms beyond 3NF. Students will not be able to critically evaluate alternative designs, or make informed choices about levels of normalization, if they do not understand the principles upon which design decisions have been based. *Functional dependencies* [1] are also important for a proper understanding of the concepts of *candidate keys*, *superkeys*, constraints, and the theoretical foundations of relational database systems.

The PROLOG programs that we have developed are able to lead a student through the process of decomposing relations to satisfy the requirements of a particular normal form for database design. The tool is able to explain the steps that are being taken to generate a decomposition, and can thus provide a solution to a given decomposition problem whilst also providing an explanation about how it was achieved. A fairly simple interface program written in Java allows the students to enter un-normalized relations and go through the steps of normalizing the relations to a "higher" normal form without having to interact directly with PROLOG themselves. In this way, students are able to compare not only a solution to a problem with their own, but to see a whole method for the problem worked out, step by step, and to call on a textual explanation facility at any point at which they do not understand the processes that have taken place. Because our tool enables students to freely select inputs, it enables them to work on problems of their own devising, at a level of difficulty exactly appropriate to their own level of understanding.

The remainder of this paper is organized as follows. In Section 2, a number of preliminary issues are discussed. In Section 3, the implementation of the teaching tool is described. In Section 4, a sample user session is described. In Section 5, the evaluation of the software is considered. Finally, conclusions and suggestions for further work are given in Section 6.

We assume that the reader has a knowledge of the basic notions and notations that are typically used in discussions on dependency theory; otherwise, we suggest [1] for all necessary background information.

2 Preliminaries

Our approach to developing our learning tool for database design initially involved us adopting a *phenomenographic* method [5] for information gathering on students' understanding of concepts in dependency theory. By conducting 'dialogue' sessions with students we identified the strategies students used to understand the basic concepts. From our review of the notes taken at the dialogue sessions, we were able to develop a prototype system for supporting students in learning about dependency theory.

As the software evolved, we made increasing use of Gagne's *event-based model of instruction* [6] to decide what material a user of the tool should be offered and the order in which information ought to be presented to a learner. Thus, different levels of learning guidance are available to meet the requirements of an individual student, and learning takes place in a student-centred, interactive way.

In overview, our learning tool includes implementations of the following algorithms. We use Ullman's FD-closure algorithm [1]; we use Beeri and Honeyman's algorithm [7] for checking dependency-preservation after decomposition; we use Loizou and Thanisch's approach [8] for checking for a lossless-join decomposition; we use Ullman's method for finding a minimal cover for a set of FDs [1]; we use Gottlob's method [9] for computing a cover for the projection of a set of FDs onto a subschema of the decomposition; and we use Luchessi and Osborn's key finding algorithm [10] to identify candidate keys. For the decomposition algorithms, we used the proposal in Ullman [1] for generating 3NF schemes, and Tsou and Fischer's approach [11] for generating BCNF schemes.

In our approach, an n-ary *relational scheme* of the form

$$R(A_1, \ldots, A_n)$$

where R is the name of the relation and $A_1, \ldots, A_n$ is a set of attributes is represented in our PROLOG implementation by using a list: $[A_1, \ldots, A_n]$.

Moreover, given a functional dependency of the form:

$$A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n$$

where A_i ($i \in \{1, .., m\}$) and B_j ($i \in \{1, .., n\}$) are attributes, we use pairs of lists and define an operator for $\rightarrow$, to wit:

$$[A_1, \ldots, A_m] \rightarrow [B_1, \ldots, B_n]$$

3 Implementation

Our implementation of the database design tool includes a GUI that sits on top of the PROLOG implementation of the database design algorithms.

The interface is intended to be simple to use; it is menu-based and all data that is entered is case-insensitive. Users are prompted throughout a session for the correct data to enter, and can return to the main menu at any time. It is possible to enter multiple schemas, save them, and return to them later within a session. Sessions can also be retained in a file, and can therefore be suspended and resumed.

The interface program is written in Java. Java has many advantages for this kind of application. It is widely used, it has comprehensive Internet support (see below), and it is easy to access applications written in a variety of other languages (through the *Java Native Interface (JNI)* mechanism).

We use XSB-Prolog [12] to implement the main logic programs that implement the database design algorithms. XSB runs on a number of platforms and offers excellent performance that has been demonstrated to be far superior to that of traditional Prolog-based systems [13].

Calls to XSB from the interface program are handled by the YAJXB [14] package. YAJXB makes use of Java's JNI mechanism to invoke methods in the C interface library package supplied by XSB. It also handles all of the data type conversions that are needed when passing data between C and Prolog-based applications. YAJXB effectively provides all the functionality of the C package within a Java environment.

Although we have used YAJXB in our implementation, we note that a number of alternative options exist. Amongst the options that we considered were Interprolog, a Java-based Prolog interpreter (e.g., JavaLog), or a Sockets-based, direct communication approach. Unfortunately, each of these approaches has its own distinct drawbacks when compared with the approach that we adopted. Interprolog does work with XSB, so we could still take advantage of the latter's performance capabilities. However, Interprolog is primarily a Windows-based application. All of our development was done on a Sun Sparc/Solaris system; YAJXB, though primarily configured for Linux, compiles easily on Solaris. JavaLog

was discounted because we felt that it did not offer sufficient flexibility compared with XSB. Finally, using sockets would give us a less portable application because it would involve considerably more application-specific coding. Overall, we felt that the straightforwardness of the YAJXB interface makes it preferable to the Interprolog approach so far as interfacing with XSB is concerned. Moreover, XSB's highly developed status and excellent performance make it more desirable in this context than a Java/Prolog hybrid.

The C library allows the full functionality of XSB to be used. A variety of methods for passing Prolog-style goal clauses to XSB exists. However, we generally found that the string method worked well. This method involves constructing a string σ in a Java String type variable, and using the *xsb_command_string* function (or similar) to pass σ to XSB. This approach allows any string that could be entered as a command when using XSB interactively to be passed to XSB by the interface program. YAJXB creates an interface object; the precise method of doing this is a call like:

i=core.xsb_command_string (command.toString());

where the assignment, as one would expect, handles the returned error code. Variations on this method allow for the return of data where relevant.

4 A Learning Session

Users invoke the software by using the Java JRE. On invoking the software, the system will respond with the opening menu:

MAIN MENU

1. Enter a Schema
2. Help
3. Exit

Enter Choice (1-3):

The "help" option gives some general guidance on how to use the system; the "exit" option terminates the program. Having invoked the system, the user will normally enter a schema. The system will first prompt the user to enter the names of the attributes:

Enter attribute names, using spaces to separate them.
Names must be single characters or strings:

If the required schema attributes are (a, b, c, d) then the user will respond, to the request for input, with something like:

a b c d

The next step is for the functional dependencies to be entered. The user is first prompted to enter a determinant, then its dependent attributes. The process will be repeated for each determinant. The system loops around these input processes until a blank line is entered: for each determinant the user is repeatedly prompted to enter another dependent attribute until a blank line is entered; the user is then invited to enter another determinant, and this process in turn repeats until a blank line is entered. For example, for the functional dependency $a \rightarrow bc$ we have:

Enter a determinant
If multivalued, use spaces as separators: a

Enter a dependent attribute
If multivalued, use spaces as separators: b

Enter a dependent attribute
If multivalued, use spaces as separators: c

Enter another dependent attribute
(return to end):

...

When the process of describing functional dependencies is complete, the system will respond with a display of the information entered and another menu. All functional dependencies are displayed in *right reduced form* i.e., with a single set of attributes on the right-hand side of an FD. For example:

Your schema has attributes: [a,b,c,d]

and FDs: [a]→[b], [a]→[c], [c]→[d]

CHOOSE AN OPTION:

1. 3NF decomposition
2. BCNF decomposition
3. Help
4. Exit

Enter Choice (1-4):

The "exit" option returns the user to the previous menu; the "help" option gives some general information about the decomposition process. The example that follows gives some sample output if 3NF decomposition is selected from the menu:

```
Finding a minimal cover
At each step, enter ? for help or CR to continue...
...checking right reduction
...checking left irreducibility
...checking redundant FDs
Decomposing...
...checking lossless join property

The following 3NF subschemata
give the dependency preserving
decomposition of your schema:

t_1
     [a,b,c]
     one key: [a]
t_2
     [c,d]
     one key: [c]
```

Having generated the 3NF decomposition, the user can then return to the main menu, and either enter another schema, or exit the system.

5 Evaluating The Software

Our database design teaching and learning tool has been formatively evaluated. For the formative evaluation, we sought comments from several colleagues involved in teaching database management at the University of Westminster; these

were our "expert reviewers" [15]). We additionally worked with a small group of post-graduate students who were learning about dependency theory at the time at which they used the software; these students tested the software during tutorials over two consecutive weeks, and in several additional sessions. A number of suggestions made by the expert reviewers and the volunteer students were used to make minor modifications to our initial design e.g., modifications of the interface.

We then conducted a program of small-group testing with some final year undergraduate students who were also studying dependency theory as part of a database design module. We gathered feedback about the software by using observations and informal "interviews". This involved one of the authors sitting with the students and asking them to articulate their feelings about the software, and asking the students to complete a questionnaire at the end of the trial period, which asked them how, in general, they had found the software to use, and how they felt it compared with the traditional textbook alternative.

The students all reported that the software was useful in terms of helping to develop their understanding of dependency theory, and all agreed that the facility for testing their own solutions to normalization problems was motivating to use and important in developing understanding. They were unanimous in concluding that the tool was a major improvement, in terms of carrying out practical exercises, over the textbook.

It is not perhaps surprising that the overall feedback was so positive. Using something new is always more interesting and students like to use computers. Because of our desire to get the feedback, the students probably found the tutorials relating to the dependency theory material a more positive experience than those provided to support the rest of the module (the use of a couple of volunteer helpers from the earlier post-graduate test group resulted in a much higher staff-student ratio than usual).

6 Conclusions and Further Work

We have developed a teaching and learning tool for helping university students to learn some aspects of dependency theory. The results of discussions with the students who have used the software suggest that the tool is of value to students learning about dependency theory. However, much more extensive testing of the software will be necessary (e.g., a summative evaluation) before any firm conclusions can be drawn about its educational value.

There are several ways in which the tool could be further developed. In particular, we plan to produce a web-based interface, which will make the tool both easier to use, and more widely accessible. It would additionally enable us to

improve the availability of the explanation facility, which could be read in pop-up help windows at all stages of the normalization process. We also intend to develop our tool to assist students in their learning of multivalued and join dependencies and the normal forms that are associated with these types of dependencies.

7 References

[1] J. Ullman, Principles of Database and Knowledge-base Systems, Computer Science Press, 1989.

[2] Yazdani, M., New Horizons in Educational Computing, Chichester: Ellis Horwood, 1983.

[3] Nichol, J., Briggs, J., and Dean, J., Prolog, Children and Students, London: Kogan-Page, 1988.

[4] B. Byrne, Top Down Approaches to Database Design Tend to Produce Fully Normalised Designs Anyway, Proceedings of TLAD, 2003.

[5] F. Marton and P. Ramsden, What does it take to improve learning?, Improving Learning: New Perspectives, Kogan Page, 1988.

[6] R. M. Gagne, The Conditions of Learning, Holt, Reinhart and Winston, 1970.

[7] C. Beeri and P. Honeyman, Preserving Functional Dependencies, SIAM Journal of Computing, 10(3), 1981.

[8] G. Loizou and P. Thanisch, Testing a Dependency-preserving Decomposition for Losslessness, Information Systems, 8(1), 1983.

[9] G. Gottlob, Computing Covers for Embedded Functional Dependencies, PODS, 1987.

[10] C. Lucchesi and S. Osborn, Candidate Keys for Relations, Journal of Computer and System Sciences, 17(2), 1978.

[11] D. Tsou and P. Fischer, Decomposition of a Relation Scheme into Boyce-Codd Normal Form, SIGACT News 14(3), 1982.

[12] K Sagonas, T. Swift, D. Warren, J. Freire and P. Rao., The XSB System Version 2.0, Programmer's Manual, 1999.

[13] K Sagonas, T. Swift, and D. Warren., XSB as an Efficient Deductive Database Engine, ACM SIGMOD Proceedings, 1994.

[14] S. Decker, Yet Another Java XSB Bridge, http://www-db.stanford.edu/%7Estefan/rdf/yajxb/

[15] M. Tessmer, Planning and Conducting Formative Evaluations, Kogan-Page, 1993.

A Database Transaction Scheduling Tool in Prolog

Steve Barker
King's College, London, UK
steve@dcs.kcl.ac.uk

Paul Douglas
University of Westminster, London, UK
P.Douglas@wmin.ac.uk

Abstract

In this paper, we describe an item of "intelligent" educational software that is intended to help students taking university computer science courses to understand the fundamentals of transaction scheduling. The software, implemented in PROLOG, empowers students to construct their own learning environment and is able to provide tailored forms of feedback to different types of learner. We describe the development and evaluation of the software, and we present details of the analysis of the results of our investigation into the effectiveness of the software as a teaching and learning tool. Our results suggest that our learning tool provides students with a different and valuable type of learning experience, which traditional methods do not provide.

1 Introduction

In this paper, we describe an item of educational software that we have developed and used to help us to teach certain key notions from the realms of database transaction processing to undergraduate computer science students. More specifically, the software is an educational tool that is intended to "intelligently" assist computer science students in developing their understanding of CRAS property satisfaction [1]. In this context, "intelligently" may be interpreted as an ability to respond to a student's self-selected input by detecting and explaining his/her errors to them or confirming that his/her understanding is correct.

Ours is one of the first pieces of courseware to provide students with help in understanding the basic notions of database transaction processing and, to the best of our knowledge, is the first piece of software that is specifically intended for helping students to learn about the CRAS properties. The software provides students with a tutorial aid that is able to respond to questions about CRAS

property satisfaction in the same way that an "expert tutor" might; it enables a student to investigate the CRAS properties at his/her leisure and enables teaching staff to use tutorial sessions to answer any "non-standard" questions that students might have. This tool is also important because it provides students with a learning experience that no textbook can provide. More specifically, the software encourages students to learn about the CRAS properties by making and testing hypotheses. This approach appears to be the most natural way for students to learn about the CRAS properties (certainly it is the approach they naturally adopt). The traditional, text-based method that we have previously used to teach material on CRAS property satisfaction does not support learning by hypothesis formulation.

The rest of the paper is organized in the following way. Section 2 provides a brief introduction to the CRAS properties. In Section 3, some key features of the software are outlined. In Section 4, the main results produced from the evaluation of the software are described and discussed. In Section 5, some conclusions are drawn, and suggestions are made for further work.

We assume that the reader has a basic knowledge of database transaction processing; otherwise, we suggest [2] for introductory material.

2 The CRAS Properties

The CRAS properties are criteria that should be satisfied by a *schedule*.

Definition 2.1 A schedule σ over a set of transactions $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ is a strict partial order $(\mathcal{T}, <)$ (where $<$ is an "earlier than" relation) with the following properties (where o_i is an operation performed by transaction t_i and the allowed operations are r (read) and w (write)):

1. $\forall o_i \in \sigma$ (where $o \in \{r, w\}$), $\exists t_i \in \mathcal{T}$ such that $o_i \in t_i$;

2. If o_i and o_j are operations in $t_i \in \mathcal{T}$ and $o_i < o_j$ holds in t_i then $o_i < o_j$ holds in σ;

3. For any two conflicting operations o_i and o_j in σ, $o_i < o_j$ xor $o_j < o_i$.

Remark 2.1 Our definition of a schedule implies that no duplicate operations on a data item are possible, and assumes that a single CPU is used in transaction processing. If parallel processing is possible then $<$ can be replaced by $\leq$ in Definition 2.1.

Unfortunately, certain interleavings of the operations from different transactions in a schedule can cause anomalous behaviours (which cause the integrity of the data in a database to be compromised) and can raise a number of practical difficulties. For instance, it is possible for the value of a data item which has been updated by one transaction t_i in a schedule to be overwritten by another transaction t_j before t_i's update is performed on the database (this phenomenon is usually referred to as the *lost update problem* [2]). The principal sources of problems that arise when concurrently processing transactions are *conflicting operations* and *read froms* that involve reading uncommitted data.

Definition 2.2 Two operations o_i and o_j in a schedule σ are a conflicting pair (or are in conflict) iff the following conditions hold:

- o_i and o_j are operations on a common data item;
- at least one of o_i and o_j is a write operation.

Definition 2.3 For each data item x, if (i) $w_i(x) < r_j(x)$ holds in a schedule, (ii) t_i does not abort before $r_j(x)$, and (iii) every transaction (if any) that writes x between $w_i(x)$ and $r_j(x)$ aborts before $r_j(x)$ then t_j reads from t_i.

The CRAS properties help to solve certain problems, that arise as a consequence of interleaving of operations, by imposing certain constraints on the order in which operations are performed in a schedule. By ensuring that these constraints are satisfied, a database management system is guaranteed to produce schedules that are free of a class of potential problems that may violate the integrity of the data contained in a database. Moreover, satisfaction of the CRAS properties permits a DBMS to be configured to optimize the performance of transaction management.

The CRAS properties are: conflict serializability, recoverability, avoids cascading aborts and strictness. These properties are defined formally below. In these definitions, t_i and t_j denote arbitrary transactions, $T(\sigma)$ denotes an arbitrary schedule σ defined on a set of transactions T, r_i, w_i, a_i and c_i are respectively read, write, abort and commit operations by transaction t_i, $\rightarrow$ is "implication", $\wedge$ is 'and', $\vee$ is 'or', $\neg$ is negation, and $<$ denotes the "earlier than" relationship between operations.

Definition 2.4 A schedule σ on a set of transactions T is conflict serializable iff the following holds:

$$\forall t_i, t_j \in T(\sigma)\ conflict(t_i, t_j) \rightarrow \neg conflict(t_j, t_i)$$

where conflict is defined thus:

$$\forall t_i, t_j \in T(\sigma)\ r_i(x) < w_j(x) \rightarrow conflict(t_i, t_j)$$

$$\forall t_i, t_j \in T(\sigma)\ w_j(x) < r_i(x) \rightarrow conflict(t_j, t_i)$$

$$\forall t_i, t_j \in T(\sigma)\ w_i(x) < w_j(x) \rightarrow conflict(t_i, t_j).$$

Definition 2.5 A schedule σ on a set of transactions T is recoverable iff the following holds:

$$\forall t_i, t_j \in T(\sigma)\ read\_from(t_i, t_j) \rightarrow c_j \in \sigma\ \wedge\ c_j < c_i.$$

Definition 2.6 A schedule σ on a set of transactions T avoids cascading aborts iff the following holds:

$$\forall t_i, t_j \in T(\sigma)\ read\_from(t_i, t_j) \rightarrow c_j < r_i(x) \vee a_j < r_i(x).$$

Definition 2.7 A schedule σ on a set of transactions T is strict iff the following holds:

$$\begin{aligned}\forall t_i, t_j \in T(\sigma)\ w_j(x) < r_i(x) \vee w_j(x) < w_i(x) \rightarrow \\ a_j < r_i(x) \vee c_j < r_i(x) \vee \\ a_j < w_i(x) \vee c_j < w_i(x).\end{aligned}$$

Definition 2.8 The auxiliary predicate read_from is defined thus:

$$\begin{aligned}\forall t_i, t_j \in T(\sigma)\ \exists x[read\_from(t_i, t_j) \leftarrow\ w_j(x) < r_i(x) \wedge \neg(a_j < r_i(x)) \wedge \\ [\forall t_k \in T(\sigma)\ w_j(x) < w_k(x) < r_i(x) \\ \rightarrow a_k < r_i(x)]].\end{aligned}$$

Conflict serializability is the principal schedule correctness criterion that is used in practice by DBMS to avoid problems of inconsistent updating that arise during concurrent transaction execution. The recoverability criterion must be satisfied in order to preserve the semantics of commit operations in schedules. The ACA condition is a practical criterion that, if satisfied, reduces the amount of work that needs to be performed to recover from the effects of failed transactions. As the name suggests, satisfying the avoiding cascading aborts condition ensures that if a transaction aborts then it does not cause a chain (or cascade) of aborting

transactions to arise. That is, the failure of one transaction, t_1 (say), does not cause another transaction, t_2 (say), to fail which causes another transaction t_3 (say) to fail and so on. Strictness enables a particularly efficient method to be employed to manage aborted transactions i.e., by reinstalling *before images*.

3 Some Key Features of the Software

Our teaching tool enables students to test any syntactically correct schedule that they choose as input to the system. Students also have complete freedom to choose to investigate the satisfaction of any of the CRAS properties by these schedules.

The software that implements the system is written in PROLOG [3]. PROLOG has been widely used for implementing items of educational software (see, for example, [4] and [5]) and is appropriate for developing applications, like ours, which require that some form of "intelligence" be captured. The fact that the rules that define the CRAS properties can be directly translated into PROLOG's rule-based language was another reason for choosing the latter for the implementation of the software.

Our design of the software has been influenced by Gagne's work [6]. Gagne's event-based model of instruction helped us to decide what an individual learner ought to be offered and the order in which information ought to be presented to them. Following Gagne's suggestions, when students use the software they are reminded what the learning task to be performed is, and what it is they are supposed to be able to do once the learning task has been completed. Prominence is given to the distinctive features that need to be learned, different levels of learning guidance are supported for different types of learners, informative feedback is given, and learning takes place in a student-centred, interactive way, but with support available to students as and when they need it.

When engaging with the software, a user enters a schedule and selects a CRAS property to evaluate with respect to the schedule. The schedule is displayed to the user who may then pose queries on the schedule to test it for satisfaction of CRAS properties with respect to the set of axioms $\mathcal{A}$ that defines these properties and the auxiliary predicates in terms of which the CRAS properties are defined (see Definitions 2.4-2.8). The axioms in $\mathcal{A}$ are converted into PROLOG code for implementation.

Each operation in a schedule may be represented by a 4-tuple, (o,t_j,i,t_s). Here, o denotes an operation (i.e. read or write), t_j denotes a transaction performing the operation, i denotes the data item read or written by t_j, and t_s is the time at which o is performed i.e., the timestamp for o. In the case where o is a commit

or an abort, the data item is *null* since these operations are not performed on a data item. In our PROLOG implementation, each 4-tuple that describes an operation is represented as a fact of the form $o(a, t_j, i, t_s)$ where $a \in \{r, w\}$. In this context, a schedule σ is a finite set of o operations, and a PROLOG program is a pair $(\mathcal{A}^P, \sigma)$ where $\mathcal{A}^P$ is the PROLOG form of $\mathcal{A}$.

Example 3.1 The representation of a conflicting pair (N, M) of transactions may be expressed in PROLOG, thus (cf. Definition 2.2):

$$\begin{aligned} conflict(N, M) : -o(r, N, Y, T1), \\ o(w, M, Y, T2), \\ T1 < T2, N = \backslash = M. \end{aligned}$$

$$\begin{aligned} conflict(M, N) : -o(r, N, Y, T1), \\ o(w, M, Y, T2), \\ T2 < T1, N = \backslash = M. \end{aligned}$$

$$\begin{aligned} conflict(N, M) : -o(w, N, Y, T1), \\ o(w, M, Y, T2), \\ T1 < T2, N = \backslash = M. \end{aligned}$$

An example of the output for a schedule σ produced in a user session and an example of engaging with the system follows next.

Example 3.2 Suppose that a user's choice of schedule is as follows:

$$\langle w_1(x), w_1(y), r_2(u), w_1(z), w_2(z), c_1, w_2(x), r_2(y), w_2(y), c_2 \rangle$$

Then, the software displays the user schedule, thus:

Your chosen schedule was:

w,1,x,90
w,1,y,95
r,2,u,100
w,1,z,105
w,2,z,110

c,1,null,115
w,2,x,120
r,2,y,125
w,2,y,130
c,2,null,135

Thereafter, the user may evaluate CRAS properties with respect to the schedule. For example, the user may ask "is this schedule an ACA schedule?" i.e., *?-aca.*. In this case the output is "*yes*". A user can ask for an explanation of this result by posing the following query: *?-explainaca.* To which the software will respond:

> *Transactions in this schedule only read data items AFTER they have been written by transactions that have committed.*

Similarly, the query *?-st.* for the schedule above (i.e. "is this schedule strict?") is answered "*no*" by the software. If the user then poses the query *?-explainnonst.* (i.e. why is this schedule not strict?) then the software will respond with the following explanation:

> *Transaction 2 overwrites the data item z written by transaction 1 but BEFORE transaction 1 reaches its commit point.*

All CRAS property satisfaction questions are evaluated as described in the previous example, and several levels of explanation are provided by the software.[1]

4 Evaluation of the Software

We have performed a formative evaluation and a summative evaluation of our teaching tool.

In brief, the aim of the formative evaluation of the software was to provide information that would enable us to develop the software to a point at which it could be summatively evaluated. The summative evaluation was intended to help us to decide whether the software was of value in helping students to understand the details of CRAS property satisfaction; how the software compared in this respect to the standard text on CRAS property satisfaction [7]; and the extent to which each means of instruction was perceived by students to be motivating to use (or otherwise), and of value in helping them to learn about the CRAS properties.

[1] Several levels of explanation are offered in the sense that users are able to check the correctness of a query in respect of any of the CRAS properties. If the query is not correct they can attempt to correct it; alternatively, they can ask the software *why* it is not correct.

Although our study was primarily concerned with comparing the software with [7], it should be noted that we do not envisage that the two modes of instruction should be used in a mutually exclusive way. The comparison of the software and [7] in our evaluation was chosen merely to attempt to decide whether there was any evidence to suggest that the former might have some "educational value" when compared to the latter.

For the formative evaluation, comments on the software were sought from: two members of the teaching staff at the University of Westminster (the "expert reviewers"); a volunteer student from the university's MSc course in Database Systems (the one-to-one study); and a group of four volunteer students from the same course (the small-group testing). The volunteer students were randomly allocated to either the one-to-one or small-group testing (but not both). These students were learning about the CRAS properties at the time at which the formative evaluation of the software was being conducted.

In response to the feedback received from the users of the software, a number of changes were made to the software over time. For example, the software was changed from its initial form to display a user's schedule together with the explanations of the CRAS property satisfaction or violation by a schedule, and a variety of modifications were made to the front-end to enhance its appeal. The power to investigate any schedule and CRAS property was reported to be an attraction of the software, and a major advantage it had over Bernstein et al's text. The students commented that they particularly liked the fact that they could "interact" with the software, and that it "lets you decide what to learn".

The software was summatively evaluated with the cohort of 27 students at the University of Westminster who were taking the Database Administration (DBA) module as part of their BSc Computer Science degree programme in the Second Semester of the 2002/03 academic year.

A 5-point Likert scale, with 22 statements, was used to collect data about the perceptions the students had of the software and [7] as methods for facilitating understanding of the CRAS properties, and their attractiveness as learning instruments. To analyse the data produced from the Likert scale, we chose to use a t-test; the idea was to compare the matched pairs of scores produced by each respondent for the software and [7].

To analyze the information produced from the Likert scale, t-statistics were computed to compare the mean scores for the perceptions students had of the software and [7], overall and for three specific measures: perceived helpfulness as a teaching aid, motivational appeal, and the value of the on-line exercises, examples, and explanations.

The results produced from the Likert scale were very clear. In the overall measure of the two methods, the average difference in the ratings of the software

and [7] was 17.24 in favour of the software, and only one student reported that [7] was "better" than the software. The t-statistic for the comparison of average differences was 7.75. This is statistically significant at the 1% level.

Not surprisingly, given the overall results, the software was also perceived to be "better" than [7] in all three of the sub-categories of Likert scale items.

In terms of helping students to understand the CRAS properties, the average difference in scores between the software and [7] was 2.18, in favour of the software, and all but two of the students reported that the software had been of more value than [7] for helping them to learn about the CRAS properties. In the t-test comparison of the average difference in the ratings of the software and [7], the t-statistic was 3.48. This value is significant at the 2% level.

Our software was also perceived to have more motivational appeal than [7]. The average difference in the rating of the software and [7] in this case was 10.65 and every student reported that the software had been more motivating to use than [7]. The t-value of 7.53 for the comparison of average differences in ratings between the software and Bernstein et al. is significant at the 1% level.

The exercises, explanations and examples that are included in the software were almost unanimously perceived by the students to be of more value than those in [7] for helping them to understand the CRAS properties (for one student, however, the difference in their scores for the software and [7] was zero). The average difference in scores on the value of the exercises, explanations and examples was 4.41 in favour of the software. The t-statistic for the average difference was 8.45 which is (again) significant at the 1% level.

5 Conclusions and Further Work

Our software shows that a suitable tool can be developed to help computer science students to learn about the CRAS properties. The package enables students to construct their own learning environments by using a piece of courseware that is able to interpret and immediately explain a student's mistakes as well as being able to confirm it when his/her understanding is correct. As such, the software provides students with "intelligent" tutorial support for learning about the CRAS properties. The software is also based on sound principles of learning [7], is able to deal with any syntactically correct schedule, and it can be extended to accommodate any number of examples or exercises without requiring changes to the core set of rules on which the software is based.

Using the software enables students to: choose to investigate any of the CRAS properties using schedules of their own choice; make hypotheses about schedules satisfying the CRAS properties; test these hypotheses; and explore the conse-

quences of CRAS property satisfaction by manipulating the operations included in a schedule. As such, the software empowers students to take control of their own learning, they can learn at their own preferred pace, they can investigate their own misunderstandings and reinforce their own understanding of the CRAS properties. The fact that the software encourages students to "learn by hypothesizing" is particularly important because this is the approach students naturally adopt to learn about the CRAS properties. Using textbooks does not enable this type of learning to be supported, and can only offer students a limited number of schedules and examples of CRAS property satisfaction; textbooks cannot provide interactive feedback to students investigating schedules and schedule properties of their own choosing. Unlike their human tutors, the software has the additional attraction of providing students with tutorial support in their learning of the CRAS properties whenever they require it.

The results produced by our summative assessment of the software indicate that it was perceived by our students to be superior to [7] in a number of respects. However, more work will be required on the issue of student perceptions of the software and [7] before any definite conclusions may be drawn about their relative value. Our experience of conducting this study has also revealed that some students have a tendency to believe that a piece of educational software has to invariably be better than a text; these students regard the former as being "the future" whilst the latter is viewed as being distinctly *passé*. We intend to investigate the implications of this attitude in the near future. Moreover, while the Likert scale test revealed that the software was perceived to be helpful to students learning about the CRAS properties, further research is required to try to establish *why* this is the case.

A number of extensions to the software are possible. For example, it could be extended to permit the investigation of other types of schedule properties (e.g. *rigour* [8]) and with minor modifications the software can be used as a tutorial aid for learning about *optimistic concurrency control* [9]. We would also like to investigate the addition of a more "friendly" user environment.

6 References

[1] Barker, S., Proving Properties of Schedules, Proc. IEEE Workshop on Knowledge and Data Engineering, 174-180, 1998.

[2] Gray, J. and Reuter, A. (1993) *Transaction Processing: Concepts and Techniques*, San Mateo, CA: Morgan Kaufmann.

[3] Bratko, I. (1986) *PROLOG Programming for Artificial Intelligence*, Reading, MA: Addison-Wesley.

[4] Yazdani, M. (1983) *New Horizons in Educational Computing*, Chichester: Ellis Horwood.

[5] Nichol, J., Briggs, J., and Dean, J. (1988) *Prolog, Children and Students*, London: Kogan-Page.

[6] Gagne, R. M. (1970) *The Conditions of Learning*, NY: Holt, Reinhart and Winston.

[7] Bernstein, P., Goodman, N., and Hadzilacos, V. (1987) *Concurrency Control and Recovery in Database Systems*, Menlo Park, CA: Addison Wesley.

[8] Briebart, Y., Georgakopoulos, D., Rusinkiewicz, M., and Silbershatz, A. (1991) On Rigorous Transaction Scheduling, *IEEE Transactions on Software Engineering*, 17, 954-960.

[9] El-Masri, R. and Navathe, S. (2003) *Fundamentals of Database Systems*, Redwood City, CA: Benjamin Cummings.

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

http://www.ep.liu.se/