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.