PrettyProlog: A Java Interpreter and Visualizer of Prolog Programs

Poster Paper

Alessio Stalla Viviana Mascardi Maurizio Martelli

DISI - Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy.
alessiostalla@gmail.com, {mascardi, martelli}@disi.unige.it

Abstract. Many years of teaching demonstrated that one of the hardest concepts for students facing Prolog is the construction of the SLD tree. For this reason one student and the teachers of the Artificial Intelligence Course held at the Computer Science Department, University of Genova, developed PrettyProlog: an interpreter for a subset of Prolog, written in Java, born for didactic use. PrettyProlog features a GUI which allows the user to visualize the inner functioning of the interpreter, namely the construction of stack and SLD tree, and can be used to implement and graphically trace sophisticated programs involving cut, negation as failure, meta-programming.

1 Introduction

PrettyProlog originates from the collaboration between one student and the teachers of the Artificial Intelligence Course held at the Computer Science Department, University of Genova, for providing concrete answers to demands raised by Prolog novices. Many years of teaching demonstrated that one of the hardest concepts for students facing Prolog, is the construction of the SLD tree. A common mistake that students make is to draw one child for each atom in a goal, instead of one child for each clause whose head unifies with the selected atom. The illegitimate children consist, in the best case, in the body of the first clause usable for the corresponding atom, but more bizarre solutions are proposed almost any year. However, since SLD trees are usually large and would require a careful organisation of the space for being properly represented, drawing them on-the-fly on a blackboard, may bother experienced teachers too: typos are just behind the corner. The way bindings of variables are propagated is also difficult to understand, and the link between the SLD tree, which is something static, and the dynamics of the Prolog interpreter and of the Prolog Stack, remains often obscure.

For these reasons, we developed PrettyProlog: an interpreter for Prolog written in Java, born for didactic use, supporting list management, cut, negation as failure and meta-programming. Currently, PrettyProlog does not support the "is" predicate. PrettyProlog is freely available under the General Public License and can be downloaded from `http://alessiostalla.altervista.org/software/prettyprolog/index.php`. It features a GUI which allows the user to visualize the inner functioning of the interpreter, namely the construction of stack and SLD

tree. It has been developed from scratch, without reusing any existing Prolog implementation. Several open-source prologs were examined including TuProlog, `http://www.alice.unibo.it/xwiki/bin/view/Tuprolog/`, and many others. However, either they were too simple, making it worthless to spend time in carefully studying them in order to extend them, or they were too complex, targeted to professional users and containing many additional features and optimizations that made them quite complicated and unsuitable for our didactic purpose. Also, most of them did not manage the stack in an explicit way, due to efficiency concerns. Since none satisfied completely the need for a stack-based, easily expandable and customizable Prolog system, we designed and developed PrettyProlog as a new and really "open" piece of software, in terms of customization and re-usability.

PrettyProlog has been designed to be simple, modular, and easily expandable. For example, the core engine has an event-driven interface that makes it easy to deeply control the solving process without sub-classing the engine itself or re-writing engine methods. Also, as a design choice, simplicity has been favoured against strong encapsulation when needed. So for example, nothing prevents the programmer to corrupt the engine's stack during goal solving. The documentation explicitly states when it is unsafe to modify a data structure returned by some method.

This paper is organized in the following way: Section 2 describes the GUI offered by PrettyProlog for didactic use (stack and SLD viewers), Section 3 describes related and future work, and concludes.

2 Teaching with PrettyProlog: The Stack and SLD Tree Viewers

The PrettyProlog GUI contains various panels: the Theory panel shows the current theory. When PrettyProlog starts, this panel is empty. To load a theory, the File→Load theory menu item should be used. When PrettyProlog is used as an applet, it cannot access the local file system for security reasons. In this case a window appears where the theory can be typed in or pasted from another program. Otherwise, when used as a standalone application, the file system can be browsed and the file containing the theory can be selected. Once the theory has been loaded, it is possible to solve goals by typing them into the input field in the lower part of the window and pressing Enter or clicking Solve. If the step-by-step mode is enabled, it will be necessary to press Enter again, or click Continue, every time a frame is pushed or popped on/off the stack. Else, PrettyProlog's engine will allow the user to continue only after it has found a solution to the goal. It is also possible to stop solving the goal using the Stop button. If the goal contains free variables, it will be possible to continue until there are no more substitutions that make it true. If the goal is ground, though, as soon as a solution is found the solving process is stopped.

The SLD tree viewer panel shows as a tree the series of steps the engine has performed. Each branch represents the selection of a clause from the theory, while leaves are either solutions or dead ends, i.e. goals that could not be solved. Near each node there is the substitution that was valid at that point. Also, the SLD tree shows which frames are removed from the stack as the effect of a cut, by printing them with a different font and icon.

PrettyProlog implements basic data types (integer and real numbers, lists, strings), and offers meta-programming facilities that, combined with the "cut" predicate, make the definition of negation as failure possible. Thanks to these features, sophisticated programs may be implemented.

Figure 1 shows the SLD tree of a Prolog program that implements a classical instance of a search problem: that of moving from a city in Romania (Arad, in our case) to Bucharest [4, Chapter 3]. A Stack Viewer is also available; we cannot show it for space constraints. We implemented a depth first search with control of cycles, as well as the auxiliary `not` and `member` predicates:

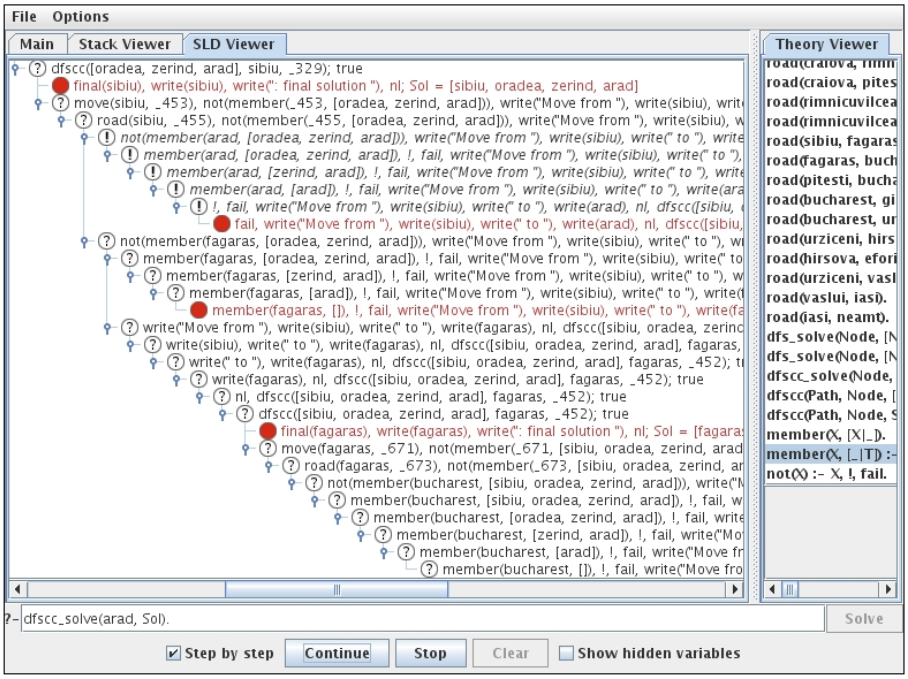

Fig. 1. SLD Viewer.

```
dfscc_solve(Node, Solution) :-
  dfscc([], Node, Solution).

dfscc(Path, Node, [Node|Path]) :-
  final(Node),
  write(Node), write(": final solution "), nl.

dfscc(Path, Node, Solution) :-
  move(Node, Node1),
```

```
  not(member(Node1, Path)),
  write("Move from"), write(Node), write("to"),  write(Node1), nl,
  dfscc([Node|Path], Node1, Solution).

member(X, [X | _]).        member(X, [_ | T]) :- member(X, T).

not(X) :- X, !, fail.      not(X).
```

The problem formulation is given by the following state graph:

```
final(bucharest).

move(St1, St2) :- road(St1, St2).

road(arad, zerind).
road(arad, timisoara).
road(sibiu, arad).
road(zerind, oradea).
road(oradea, sibiu).
road(sibiu, fagaras).
...
```

Besides showing what happens both to the stack and to the SLD tree (while it is built), PrettyProlog correctly visualizes the effect of a "cut" on the SLD tree. In the upper part of Figure 1 there are goals written in italic (from *not(member(arad, [oradea, zerind, arad])), ...* to *!, fail, write(....)*). These nodes are cut after the execution of the `!` in the first clause defining `not`, called with `member(arad, [oradea, zerind, arad]))` as argument. PrettyProlog SLD viewer keeps the cut goals for didactic purposes, but shows them in a different font to emphasise that they no longer belong to the tree. The system predicates supported by PrettyProlog, although limited, include simple predicates for input-output, such as `write` and `nl`.

The stack viewer shows each frame pushed onto the stack. When the user clicks on a frame, its content is displayed: the goal that still had to be solved at the time the frame was pushed on the stack; the substitution that is the partial solution to such goal at this point; the clause that has been used to obtain the goal; the index from where, on backtracking, the engine will search for the next clause.

When the PrettyProlog engine solves a goal step-by-step, the clause used in each resolution step is highlighted in the theory panel.

3 Related and Future Work

Research on visualization of the execution of Prolog programs has a long history (just to make some examples, [5,2,6] and many papers collected in [1]). Nevertheless, nowadays few Prolog implementations offer an on-line Stack Viewer and an SLD tree visualizer as graphical means for debugging. The open-source implementations that provide these facilities are even fewer. Among them, we acknowledge SWI-Prolog (`http://www.swi-prolog.org/`) that offers a debugging

window showing current bindings; a diagrammatic trace of the call history; and a highlighted source code listing (http://www.cs.bris.ac.uk/Teaching/Resources/COMS30106/labs/tracer.htm). No SLD tree visualization is given. SLDNF-Draw [3], downloadable from http://www.ing.unife.it/software/sldnfDraw/, is a classical meta-interpreter that executes a Prolog program and, during execution of a node, saves the Latex instructions that generate the tree. Being based on Latex, the graphical result is definitely better than ours, but, on the other side, SLDNF-Draw provides no support to run-time drawing of the tree. Also, the stack is not visualized.

On the other hand, many Java implementations of a Prolog interpreter exist, starting from W-Prolog, http://goanna.cs.rmit.edu.au/%7Ewinikoff/wp/, the first prolog interpreter written in Java, and reaching more than 20 current implementations of either Prolog interpreters written in Java, or Java/Prolog interfaces http://kaminari.scitec.kobe-u.ac.jp/logic/jprolog.html. We already explained in the Introduction the reasons why we felt the need of implementing our own Prolog interpreter in Java.

Although at a prototypical stage, PrettyProlog presents three features that, to the best of our knowledge, cannot be found together in any other Prolog implementation: 1. it provides Stack and SLD Tree visualizers; 2. it is open source; 3. it is written in Java, and fully compliant with Java ME CDC application framework.

As far as the last point is concerned, the light implementation of PrettyProlog and its compliance with the Java ME CDC application framework make it suitable for other than mere educational purposes. In particular, PrettyProlog might be run on PDAs. We are currently designing its extension for allowing engines running on interconnected PDAs to exchange knowledge and to exploit exogenous facts in successive derivations.

A further step in the development of PrettyProlog could be the adoption of either JGraph, http://www.jgraph.com/, a free Java library that allows the developer to easily draw graphs with arcs and nodes and to place the elements without intersections, or Eclipse GEF http://www.eclipse.org/gef/. In the latter case, PrettyProlog should be extended and embedded into the Eclipse IDE as a plug-in. The adoption of some existing graphic library should greatly improve the results of our SLD tree visualizer whose user-friendliness is currently limited.

References

1. M. Ducassé, A-M Emde, T. Kusalik, and J. Levy, editors. *Logic Programming Environments, ICLP'90 Preconference Workshop*. 1990. ECRC Technical Report IR-LP-31-25.
2. M. Eisenstadt and M. Brayshaw. A fine-grained account of Prolog execution for teaching and debugging. *Instructional Science*, 19(4/5):407–436, 1990.
3. M. Gavanelli. SLDNF-Draw: a visualisation tool of prolog operational semantics. In G. Fiumara, M. Marchi, and A. Provetti, editors, *CILC 2007*, 2007.
4. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Second Edition*. Prentice Hall, 2003.
5. H. Shinomi. Graphical representation and execution animation for Prolog programs. In *MIV*, pages 181–186. IEEE Computer Society, 1989.
6. D. E. Tamir, R. Ananthakrishnan, and A. Kandel. A visual debugger for pure Prolog. *Inf. Sci. Appl.*, 3(2):127–147, 1995.