A Prolog Technology Theorem Prover:
A New Exposition and Implementation in Prolog

Mark E. Stickel
Artificial Intelligence Center
SRI International
Menlo Park, California 94025

Abstract

A Prolog technology theorem prover (PTTP) is an extension of Prolog that is complete for the full first-order predicate calculus. It differs from Prolog in its use of unification with the occurs check for soundness, depth-first iterative-deepening search instead of unbounded depth-first search to make the search strategy complete, and the model elimination reduction rule that is added to Prolog inferences to make the inference system complete. This paper describes a new Prolog-based implementation of PTTP. It uses three compile-time transformations to translate formulas into Prolog clauses that directly execute, with the support of a few run-time predicates, the model elimination procedure with depth-first iterative-deepening search and unification with the occurs check. Its high performance exceeds that of Prolog-based PTTP interpreters, and it is more concise and readable than the earlier Lisp-based compiler, which makes it superior for expository purposes. Examples of inputs and outputs of the compile-time transformations provide an easy and quite precise way to explain how PTTP works. This Prolog-based version makes it easier to incorporate PTTP theorem-proving ideas into Prolog programs. Some suggestions are made on extensions to Prolog that could be used to improve PTTP’s performance.

1 Introduction

A Prolog technology theorem prover (PTTP) is an extension of Prolog that is complete for the full first-order predicate calculus [17]. We present here a new exposition and implementation of PTTP that uses Prolog to explain and implement PTTP.

PTTP is characterized by the use of sound unification with the occurs check where necessary, the complete model elimination inference procedure rather than just Prolog inference, and the depth-first iterative-deepening search procedure rather than unbounded depth-first search. These particular inference and search methods are used instead of other complete methods because they can be implemented using basically the same implementation ideas, including compilation, that enable Prolog’s very high inference rate. Other inference systems and search methods may explore radically different and smaller search spaces than PTTP, but PTTP’s design enables it to come closer to matching Prolog’s inference rate.

Several PTTP-like systems have been implemented:

- A Lisp-based interpreter [15].
- A Lisp-based compiler [17].

1This research was supported by the National Science Foundation under Grant CCR-8611116. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the National Science Foundation or the United States government.

2The name connotes two things: PTTP employs highly efficient Prolog technology in its implementation. It is also a technology theorem prover in the same way that TECH was a technology chess program [5], i.e., it is a "brute force" theorem prover that relies less on detailed analysis than on high-speed execution of small logical steps and whose capabilities will increase as Prolog machine technology progresses.
• F-Prolog, a Prolog-based interpreter [20].

• Expert Thinker, a commercial version of F-Prolog [13].


• SETHEO and PARTHEO, sequential and parallel Warren abstract machine implementations inspired by the connection method with input-formula preprocessing and additional inference and search strategy options [8].

Several other deduction systems developed in recent years also use features associated with PTTP, such as compiled inference operations for the full first-order predicate calculus, especially for linear strategies, and the use of depth-first iterative-deepening search in deduction. We present here a new implementation of Prolog using a Prolog-based compiler. First-order predicate calculus formulas are translated by the PTTP compiler, written in Prolog, to Prolog clauses that are compiled by the Prolog compiler and will then directly execute the PTTP inference and search procedure.

The new implementation has several advantages. First, its performance is high, although still not equal to that of the Lisp-based compiler implementation.

Second, the Prolog-based PTTP should generally produce much shorter object code than our Lisp-based compiler and compilation speed should also be improved. The Prolog clauses produced by the PTTP compiler typically will be compiled by the Prolog compiler to a concise abstract-machine target language. Our Lisp-based PTTP compiled its input to Lisp code that was then compiled to machine code rather than a Prolog abstract-machine language, so object code could be quite large and compilation time long.

The code for the Prolog-based version is also shorter and more perspicuous than that for the Lisp-based version. Modifiability is enhanced. Elements of PTTP, like logical variables and backtracking, that are basic features of Prolog had to be explicitly handled in the Lisp version of the PTTP compiler. In effect, we had to write a PTTP-to-Prolog compiler and a Prolog-to-Lisp compiler for the Lisp version; for this Prolog-based version, only the former is necessary.

The Prolog-based version is also more readily usable by those who would like to incorporate PTTP reasoning for some tasks into larger logic programs written in Prolog. Since the output of this PTTP-to-Prolog compiler is pure Prolog code, it is easy to achieve parallel execution of PTTP inference by simply executing the code on any parallel implementation of standard, sequential Prolog.

Finally, we feel that this version of PTTP in Prolog has pedagogical value. This description, and the code for the PTTP-to-Prolog compiler, explain clearly and precisely the principles of a Prolog technology theorem prover. Example inputs and outputs of the transformations used by PTTP clearly describe PTTP's operation.

We illustrate by example PTTP's recipe for transforming first-order predicate calculus formulas to Prolog clauses that, when executed, perform the complete model elimination theorem-proving procedure on the formulas.

First, first-order predicate calculus formulas are translated to Prolog clauses and their contrapositives.

The recipe then specifies application of

• A compile-time transformation for sound unification that linearizes clause heads and moves unification operations that require the occurs check into the body of the clause where they are performed by a new predicate that performs sound unification with the occurs check.

• A compile-time transformation for complete depth-bounded search that adds extra arguments for the input and output depth bounds to each predicate and adds depth-bound test and decrement operations to the clause bodies.