Efficient Prolog: a practical tutorial

Michael A. Covington
University of Georgia, Athens, Georgia 30602, USA

Abstract. This tutorial presents twelve guidelines for making Prolog programs efficient. Topics covered include procedural versus declarative programming, internal representation of data, first-argument indexing, cuts, and ways to avoid CONSing.

Introduction

Prolog is often said to be inefficient. This is partly because of the poor performance of early experimental implementations, but another problem is that some programmers use Prolog inefficiently.

Properly used, Prolog is as fast as any other language with comparable power. It is certainly as fast as Lisp, if not faster (Warren & Pereira, 1977). There are those who rewrite Prolog programs in C 'for speed', but this is tantamount to boasting, 'I can implement the core of Prolog better than a professional Prolog implementor'.

This paper is a tutorial on making Prolog programs efficient. The intended audience comprises Prolog programmers who can get the computer to do what they want, but wonder if they are doing things the best way. The twelve guidelines given here are general and go well beyond the implementation-specific advice normally given in manuals. For more advice along the same lines, see O'Keefe (1990); for an introduction to Prolog with many sample applications, see Covington et al. (1988).1

1 Think procedurally as well as declaratively

Prolog is usually described as a declarative or non-procedural language. This is a half-truth. It would be better to say that most Prolog clauses can be read two ways: as declarative statements of information and as procedures for using that information. For instance, the clause

\[
\text{in}(X, \text{usa}) :- \text{in}(X, \text{georgia}).
\]

means both 'X is in the USA if X is in Georgia' and 'To prove that X is in the USA, prove that X is in Georgia'.

Prolog is not alone in this regard. The C statement \(x = y + z\) can be read both declaratively as the equation \(x = y + z\) and procedurally as the instructions \text{LOAD} \ y, \text{ADD} \ y, \text{STORE} \ x.
ADD Z, STORE X. Of course declarative readings pervade Prolog to a far greater extent than C or Pascal.

Sometimes the declarative and procedural readings conflict. C lets you utter the mathematical absurdity \( x = x + 1 \). More subtly, the C statements

\[
a = (b+c)+d  \\
a = b+(c+d)
\]

look mathematically equivalent, but they give profoundly different results when \( b = 10000000, c = 10000000, \) and \( d = 0.0000012345 \).

Analogous things happen in Prolog, but they involve logic as well as arithmetic. To take a familiar example, the clause

\[
\text{ancestor}(A, C) :\leftarrow \text{ancestor}(A, B), \text{ancestor}(B, C).
\]

is part of a logically correct definition of \( \text{ancestor} \), but it can cause an endless loop when Prolog interprets it procedurally.

The loop arises because, when \( B \) and \( C \) are both unknown, the goal \( \text{ancestor}(A, B) \) on the right is no different from \( \text{ancestor}(A, C) \) on the left. The clause simply calls itself with effectively the same arguments, making no progress toward a proof. But if the clause is rewritten as

\[
\text{ancestor}(A, C) :\leftarrow \text{parent}(A, B), \text{ancestor}(B, C).
\]

there is no loop because \( \text{ancestor} \) can no longer call itself with the same arguments.

The moral is that to use Prolog effectively, one must understand not only the declarative reading of the program but also the procedures that the computer will follow when executing it. The limitations of Prolog's built-in proof procedures are not flaws in the implementations; they are deliberate compromises between logical thoroughness and efficient search.

2 Narrow the search

Searching takes time, and an efficient program must search efficiently. One useful strategy is to make each test fail as early as possible. If horses are common and albinos are rare, then

\[
\text{albino\_horse}(X) :\leftarrow \text{albino}(X), \text{horse}(X).
\]

can be much faster than the alternative

\[
\text{albino\_horse}(X) :\leftarrow \text{horse}(X), \text{albino}(X).
\]

because, for any given animal, it will fail sooner. If \( \text{albino}(X) \) fails, \( \text{horse}(X) \) need not be executed at all.

Many opportunities to narrow the search space are much more subtle. Consider the problem of determining whether two lists are set-equivalent — that is, whether they have exactly the same elements, though not necessarily in the same order.

Two lists are set-equivalent if and only if one of them is a permutation of the