1. Introduction

In the Computer Science literature, Prolog is usually characterized as a language for programming computers. That makes sense. Its inventors described Prolog as a programming language; and its very name is an abbreviation for PROgramming in LOGic.

Unfortunately, characterizing Prolog as primarily a programming language may be doing it a disservice. It is also an excellent knowledge representation language. In fact, an argument can be made that Prolog's main value lies not so much in programming as in knowledge representation.

The practical difference between these viewpoints can be seen by realizing that the rules in a Prolog "program" can be used in multiple ways to solve different types of problems. In this paper, we illustrate this point by presenting three real world problems that can be solved by encoding knowledge in standard Prolog and applying different interpreters.

This paper presumes familiarity with the declarative reading of Prolog programs. The full paper associated with this paper gives formal background, presents additional examples, and discusses the distinction between programming and knowledge representation in greater detail.

2. Query Evaluation - Kinship

Query Evaluation is the simplest way in which a Prolog program can be used. We start with a ruleset and a dataset and apply a query evaluation procedure to compute answers to queries. The interpreter could be a bottom-up interpreter or a top-down evaluator (like the standard Prolog interpreter) or some combination with optimization refinements such as conjunct ordering and/or tabling.

Suppose, for example, we have a dataset of kinship information like the one below. The person named art is a parent of the person named bob and the person named bea; bob is the parent of cal and cam; and bea is the parent of cat and coe.

parent(art,bob)
parent(art,bea)
parent(bob,cal)
parent(bob,cam)
parent(bea,cat)
parent(bea,coe)
The following Prolog rule defines the grandparent relation in terms of parent. A person \( x \) is the grandparent of a person \( z \) if \( x \) is the parent of a person \( y \) and \( y \) is the parent of \( z \).

\[
\text{grandparent}(X,Z) :- \text{parent}(X,Y) \& \text{parent}(Y,Z)
\]

Given this dataset and ruleset, we can apply a query evaluation procedure to compute the corresponding instance of the grandparent relation.

\[
\begin{align*}
\text{grandparent}(\text{art},\text{cal}) \\
\text{grandparent}(\text{art},\text{cam}) \\
\text{grandparent}(\text{art},\text{cat}) \\
\text{grandparent}(\text{art},\text{coe})
\end{align*}
\]

Query evaluation is the usual way in which Prolog is used. The answers are logically entailed by the data and rules, and the standard Prolog interpreter produces these answers by some form of deduction (typically SLD-resolution).

3. Constraint Satisfaction - Map Coloring

Now, consider the problem of coloring planar maps using only four colors, the idea being to assign each region a color so that no two adjacent regions are assigned the same color. A typical map is shown below. In this case, we have six regions. Some are adjacent to each other, meaning that they cannot be assigned the same color. Others are not adjacent, meaning that they can be assigned the same color.

![Map Coloring Diagram]

We can represent the basic facts of this problem as a set of ground atoms like the ones below.

\[
\begin{align*}
\text{region}(r1) & \quad \text{hue(red)} & \quad \text{next}(r1,r2) & \quad \text{next}(r2,r5) \\
\text{region}(r2) & \quad \text{hue(green)} & \quad \text{next}(r1,r3) & \quad \text{next}(r2,r6) \\
\text{region}(r3) & \quad \text{hue(blue)} & \quad \text{next}(r1,r5) & \quad \text{next}(r3,r4) \\
\text{region}(r4) & \quad \text{hue(violet)} & \quad \text{next}(r1,r6) & \quad \text{next}(r3,r6) \\
\text{region}(r5) & \quad & \quad \text{next}(r2,r3) & \quad \text{next}(r5,r6) \\
\text{region}(r6) & \quad & \quad & \quad \text{next}(r2,r4)
\end{align*}
\]

One way to codify the constraints in this problem is to define a relation \( \text{illegal} \) that is true for any assignment that violates those constraints. For example, the first rule below states that no two adjacent regions can have the same color. The second rule states that no region can have more than
one color. The last two rules state that every region must have at least one color.

\[
\begin{align*}
\text{illegal} & :\text{ next}(R_1, R_2) \& \text{color}(R_1, C) \& \text{color}(R_2, C) \\
\text{illegal} & :\text{color}(R, C_1) \& \text{color}(R, C_2) \& \text{distinct}(C_1, C_2) \\
\text{illegal} & :\text{region}(R) \& \neg \text{hascolor}(R) \\
\text{hascolor}(R) & :\text{color}(R, C)
\end{align*}
\]

Given these definitions, it is possible to determine that the dataset below is a solution to the map coloring problem.

\[
\begin{align*}
\text{color}(r_1,\text{red}) \\
\text{color}(r_2,\text{green}) \\
\text{color}(r_3,\text{blue}) \\
\text{color}(r_4,\text{red}) \\
\text{color}(r_5,\text{blue}) \\
\text{color}(r_6,\text{purple})
\end{align*}
\]

Of course, this is not the only solution. It is possible to permute the colors in various ways and still satisfy the constraints. The point of the example is that none of these solutions is logically entailed by the definitions in the problem, and the standard Prolog interpreter will not produce any answers to questions about the colors of regions (given the rules as written). However, an interpreter that is capable of abduction (as opposed to deduction) can produce answers like the one above.

### 4. Containment Testing - Insurance Portfolio Analysis

A common problem in analyzing insurance products is determining whether an insurance policy or collection of policies provides coverage for a collection of possible events.

Consider the example below. Here we see the preferences of an insuree named *joe*. In particular, he wants coverage for hospitalizations for himself in Japan and Korea.

\[
\begin{align*}
\text{covered}(Z) & :\text{patient}(Z,\text{joe}) \& \text{hospital}(Z,H) \& \text{country}(H,\text{japan}) \\
\text{covered}(Z) & :\text{patient}(Z,\text{joe}) \& \text{hospital}(Z,H) \& \text{country}(H,\text{korea})
\end{align*}
\]

Here we have the definition of the hospitalizations covered by a particular policy he is considering. The insuree and his relatives are covered anywhere in Asia.

\[
\begin{align*}
\text{covered}(Z) & :- \\
& \text{patient}(Z,P) \& \text{related}(P,\text{joe}) \& \\
& \text{hospital}(Z,H) \& \text{country}(H,C) \& \text{continent}(C,\text{asia})
\end{align*}
\]

We also have some background information. The individuals related to an insuree include himself, his spouse, and his kids. And the countries of Japan and Korea are in Asia.

\[
\begin{align*}
\text{related}(I,I) & \quad \text{continent}(\text{japan},\text{asia}) \\
\text{related}(P,I) & :- \text{spouse}(P,I) \quad \text{continent}(\text{korea},\text{asia}) \\
\text{related}(P,I) & :- \text{parent}(I,P)
\end{align*}
\]
Given this information, it is easy for use to see that the policy covers the preferences of the insuree. Like the preceding examples, we have view definitions. Unlike the case of query evaluation, we do not have a complete dataset. And, unlike the case of constraint satisfaction, we are not looking for a complete dataset. Instead, we are trying to determine whether a policy covers his needs for every complete dataset. The key to automating this determination is to use an interpreter capable of containment testing.

5. Conclusion

The facts and rules in these examples are all Prolog programs written using the same declarative semantics; but, in the various examples, the programs are processed in completely different ways. This multiplicity of uses illustrates the value of using Prolog to encode the knowledge relevant to applications rather than thinking of sets of Prolog facts and rules as programs for processing that knowledge in one specific way for all applications.

References


