Prolog as a Knowledge Representation Language
The Nature and Importance of Prolog

Michael Genesereth
Computer Science Department
Stanford University

Abstract

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 [14]; and its very name is an abbreviation for PROgrammation en LOGique (PROgramming in LOGic). Unfortunately, characterizing Prolog as primarily a *programming language* may be doing it a disservice. As argued in [6] and [5] and elsewhere, 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.

1. Introduction

In the Computer Science literature, Prolog is usually characterized as a high-level programming language. However, it also has value as a knowledge representation language. The main distinction between these two viewpoints lies in the way one thinks about the semantics of Prolog programs.

The knowledge representation point of view is purely declarative. A Prolog "program" can be viewed as simply a set of inductive definitions of higher level relations in terms of lower level relations. There is no specification for how those definitions are to be used.

The programming point of view is more procedural. Prolog programs are usually assumed to be processed by a specific algorithm (based on SLD-resolution) for a specific purpose (computing answers to queries), possibly with procedural directives to guide the process.

The practical difference between these viewpoints can be seen by realizing that the rules in a Prolog program can be effectively used in multiple ways. (1) The rules can be used to deduce answers to given queries. (In this case, the two views are effectively equivalent.) (2) The rules can also be used to abduce data that produces specified query results (constraint satisfaction). (3) The

rules can be used to compare relations for disjointness or overlap or equivalence (containment testing). (4) View definitions can be "differentiated" to produce rules for computing updates to materialized views (as suggested by Orman). (5) Definitions can be automatically "inverted" to enable query folding (for purposes of data integration). And so forth. Each of these tasks requires a different "interpreter", but *the rules are the same in all cases.* (And, incidentally, few of these tasks can be performed easily with programs written in traditional imperative programming languages.)

The point is that, by focussing on Prolog as a programming language rather than a knowledge representation language, we may be doing it a disservice. In fact, it can be argued that Prolog's *main* value lies not so much in programming as in knowledge representation. In this article, I make this argument in three stages. First of all, I explain why I think Prolog is superior to other knowledge representation formalisms, such as First-Order Logic. I then discuss the merits of having multiple interpreters for Prolog. Finally, I talk about the prospects for automatically transforming logic programs from natural but potentially expensive form into versions that execute more efficiently for specific interpreters.

2. Simplicity and Completeness

In an early paper [20], John McCarthy extolled the benefits of First Order Logic (FOL) as a framework for knowledge representation. The language of FOL provides a variety of useful linguistic features, e.g. logical operators, variables, and quantifiers. Moreover, being domain-independent, the language of FOL has no built in assumptions and thus is more general than domain-specific languages.

Unfortunately, FOL has some properties that limit its usefulness. For example, in FOL, it is not possible to define the notion of transitive closure in a way that precludes non-standard models (at least without Herbrand semantics [7]). Moreover, in defining relations, it is usually necessary to write "negation axioms" to say when those relations do *not* hold as well as positive axioms that say when those relations do hold.

One nice feature of Prolog is that it deals with these limitations in a graceful way. If one abides by a few restrictions in writing Prolog programs (e.g. safety and stratified negation) and if one uses minimal model semantics for the language (i.e. negation as failure), transitive closure can be defined precisely, and negation axioms become unnecessary.

Some might argue that this feature of Prolog is also a disadvantage. Under the conditions just mentioned, every Prolog program has a unique minimal model. This effectively prevents one from encoding incomplete information. In order to deal with this disadvantage, it might be nice to allow programmers to write rules with negations or disjunctions or existential heads. And Answer-Set Programming (ASP) [19] provides a way for programmers to blend classical negation with negation as failure.

On the other hand, in many circumstances it is often desirable to strive for complete knowledge about the application area of a program. For example, in writing specifications for runnable programs, it is desirable to know what is acceptable behavior and what is not acceptable so that a system executing the program can act with confidence. While it is possible to write complete theories in FOL, it is not always easy to determine whether or not a given theory is complete. By contrast, in Prolog (with safety and stratified negation), one knows that the theory is complete. And, when one absolutely needs to express incomplete information, ASP provides a natural extension to Prolog to express this information.

3. Multiple Interpreters

The main distinction between the view of Prolog as a *programming language* and the view of Prolog as a *knowledge representation language* lies in the way one thinks about the semantics of Prolog "programs". The KR point of view is purely *declarative*. A Prolog program can be viewed as simply a set of inductive definitions of higher level relations in terms of lower level relations. There is no regard for how those definitions are to be used. The programming point of view is more *procedural*. Prolog programs are assumed to be processed by a specific algorithm for a specific purpose (computing answers to queries), possibly with procedural directives to guide the process [10].

If one's only interest is getting answers to queries, then the two approaches are effectively equivalent. The declarative semantics specifies which answers are correct, and the Prolog interpreter computes those answers.

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 what follows, we illustrate this point by presenting three real world problems that can be solved by encoding knowledge in standard Prolog and applying different interpreters.

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 [4]) alone or in combination with optimization refinements such as conjunct ordering and/or tabling [26].

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

```
grandparent(X,Z) :- parent(X,Y) & parent(Y,Z)
```

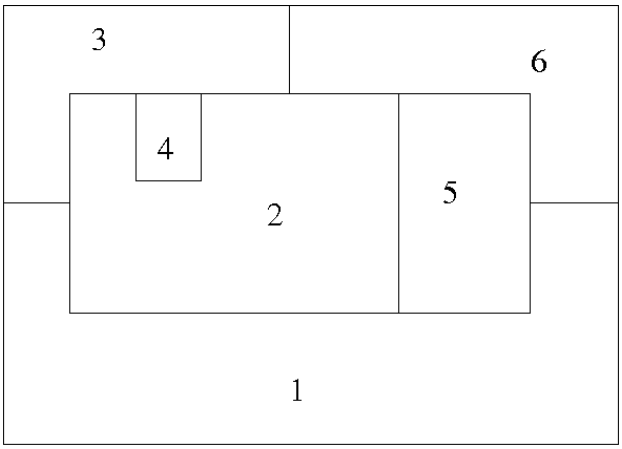
Given this dataset and ruleset, we can apply a query evaluation procedure to compute the corresponding instance of the grandparent relation.

```
grandparent(art,cal)
grandparent(art,cam)
grandparent(art,cat)
grandparent(art,coe)
```

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 [13]).

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.



We can represent the basic facts of this problem as a set of ground atoms like the ones below. We use the unary relation `region` to enumerate regions. We use `hue` to enumerate possible colors of regions. And we use the binary relation `next` to capture contiguity between regions. (Note that the sentences here capture contiguity in one direction. One might wish to include the sentences with arguments reversed to capture contiguity in the other direction as well.)

```
region(r1)    hue(red)       next(r1,r2)    next(r2,r5)
region(r2)    hue(green)     next(r1,r3)    next(r2,r6)
region(r3)    hue(blue)      next(r1,r5)    next(r3,r4)
region(r4)    hue(purple)    next(r1,r6)    next(r3,r6)
region(r5)                   next(r2,r3)    next(r5,r6)
region(r6)                   next(r2,r4)
```

One way to codify the constraints in this problem is to define a relation `illegal`, which 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.

```
illegal :- next(R1,R2) & color(R1,C) & color(R2,C)
illegal :- color(R,C1) & color(R,C2) & distinct(C1,C2)
illegal :- region(R) & ~hascolor(R)
hascolor(R) :- color(R,C)
```

Our goal in this problem is to infer a set of ground atomic sentences that characterize the color relation. Given these definitions, it is possible to determine that the dataset below is one solution to the problem. Of course, this is not the only solution. It is possible to permute the colors in various ways and still satisfy the constraints.

```
color(r1,red)
color(r2,green)
color(r3,blue)
color(r4,red)
color(r5,blue)
color(r6,purple)
```

The point of this 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) or *constraint satisfaction can* produce answers like the one above.

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. [9]

Consider the example below. Here we see the preferences of an insuree `joe`. In particular, he wants a portfolio of policies that covers him for hospitalizations in Japan or Korea. The binary relation `patient` here relates a hospitalization and the patient. The relation `hospital` relates a hospitalization and a hospital. And the `country` relation relates a hospital and a country.

```
covered(Z) :- patient(Z,joe) & hospital(Z,H) & country(H,japan)
covered(Z) :- patient(Z,joe) & hospital(Z,H) & country(H,korea)
```

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.

```
covered(Z) :-
  patient(Z,P) & related(P,joe) &
  hospital(Z,H) & country(H,C) & continent(C,asia)
```

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.

```
related(I,I)
related(P,I) :- spouse(P,I)
related(P,I) :- parent(I,P)

continent(japan,asia)
continent(korea,asia)
```

Given this information, it is easy for us to see that the policy covers the preferences of the insuree. Like the preceding examples, we have definitions of view relations. 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* [24][1] to determine whether one program produces all of the results of another program for *any* dataset, not just one particular dataset.

4. Program Transformation

One of the nice benefits of Prolog's declarative semantics is that it gives us a natural definition for the equivalence of different programs - two programs are equivalent if and only if they compute the same results. Of course, we can define the equivalence of programs with imperative semantics in similar fashion. However, it is much easier to determine the equivalence of declarative programs than to determine the equivalence of procedural programs. Moreover, there are powerful techniques for automatically transforming declarative programs into equivalent (but computationally more efficient) form and/or compiling them

into procedural programs. (Yes, equivalence testing for Prolog is undecidable in general, but it is practical in many special cases.)

As an example, consider the map coloring problem described above. As mentioned earlier, the standard Prolog interpreter is not capable of finding solutions. However, this problem can be solved by transforming the program from its natural expression as a constraint satisfaction problem into a form suitable for execution by the standard Prolog interpreter.

The version below was proposed by Pereira and Porto [22] and subsequently mentioned by McCarthy[21]. Rather than defining on regions, we start by defining `ok` as a relation on colors, viz. the pairs of colors that may be next to each other.

```
ok(red,green)  ok(green,red)    ok(blue,red)    ok(purple,red)
ok(red,blue)   ok(green,blue)   ok(blue,green)  ok(purple,green)
ok(red,purple) ok(green,purple) ok(blue,purple) ok(purple,blue)
```

In the case of the map shown above, our goal is to find six hues (one for each region of the map) such that no two adjacent regions have the same hue. We can express this goal by writing the query shown below.

```
goal(C1,C2,C3,C4,C5,C6) :-
  ok(C1,C2) & ok(C1,C3) & ok(C1,C5) & ok(C1,C6) &
  ok(C2,C3) & ok(C2,C4) & ok(C2,C5) & ok(C2,C6) &
  ok(C3,C4) & ok(C3,C6) & ok(C5,C6)
```

Given this version, we can use the standard Prolog interpreter to produce 6-tuples of hues that ensure that no two adjacent regions have the same color. Of course, in problems like this one, we usually want only one solution rather than all solutions. However, finding even one solution is such cases can be costly.

The good news it is possible to convert the formulation described earlier into this form; and in many cases this conversion can be done automatically [11]. The benefit of doing things this way is that the programmer can formalize the problem in its most natural form - as a constraint satisfaction problem - and the system can then transform into a version that can be executed by the standard Prolog interpreter.

But, wait, there's more! Given a program's declarative semantics, it is possible to rewrite the program into a form that runs even more rapidly. In our example, the program can be improved by reordering the conjuncts in the definition of goal, based on size heuristics and/or the Kempe transformation described by McCarthy [21], resulting in the version shown below.

```
goal(C1,C2,C3,C4,C5,C6) :-
  ok(C1,C2) & ok(C1,C3) & ok(C1,C6) &
  ok(C2,C3) & ok(C2,C6) & ok(C3,C6) &
  ok(C2,C4) & ok(C3,C4) & ok(C1,C5) & ok(C2,C5) &
  ok(C5,C6)
```

This version looks the same but runs faster due to the better ordering of subgoals. This is not something that necessarily matters to the programmer; but it does matter to the interpreter. If this transformation is automated, the programmer does not need to worry these details.

Similar examples can be found in the formulation of computationally complex problems such as the fibonacci function. The programmer can write the definitions in their most natural but computationally inefficient form, and the program can be automatically written in a form that can be executed efficiently, either by rewriting the fibonacci definition with a single recursion or by using a tabling interpreter [23].

The computer programming community has long recognized the value of Interactive Development Environments (IDEs) to help in developing and maintaining programs. Logical Interactive Development Environments (LIDEs) have similar value for Logic Programming. These systems can save authors work providing pre-existing ontologies and databases and knowledge bases. They provide tools for the verification, analysis, and debugging of legal codes. And they can provide technology for automatically translating to and from other formal languages. They can support languages that are more expressive than Logic Programming, e.g. FOL and beyond. They can support languages that are more human-friendly, e.g. controlled English, such as Kowalski's Logical English [18], thus making possible pseudo-natural language authoring without the drawbacks of general natural language.

More importantly, it is possible in many cases for such environments to transform programs into computationally more efficient versions, allowing the programmer to encode knowledge in its most natural form while the computer gets to execute a more efficient version. The cost of these transformations can be paid once, and the cost can be amortized over multiple uses of the transformed programs.

5. Conclusion

The facts and rules in the examples described above are all Prolog programs written using simple *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. Transformations to enhance efficiency can be applied by an interactive development environment. The upshot is that the user can code the knowledge in its most natural form and use the LIDE to find a suitable interpreter or a computationally tractable form for the program that can be executed by the standard interpreter or a variant like XSB [23].

References

[1] P. Carlson, M. Genesereth: Insurance Portfolio Management as Containment Testing, ICAIL 2023.

[2] A. K. Chandra, P. M. Merlin: Optimal implementation of conjunctive queries in relational databases. Proceeding of the 9th Annual ACM Symposium on the Theory of Computing, pp. 77-90 (1977).

[3] W. Chen, T. Swift, D. Warren, Efficient top-down computation of queries under the well-founded semantics, J. Logic Programming 24 (3) (1995) 161-201.

[4] W. F. Clocksin, C. S. Mellish: Programming in Prolog. 4th edition. New York: Springer-Verlag. 1994.

[5] B. De Cat, B. Bogaerts, M. Bruynooghe, G. Janssens, M. Denecker: Predicate logic as a modeling language: the IDP system. Declarative Logic Programming 2018: 279-323.

[6] M. Gelfond, N. Leone: Logic programming and knowledge representation. Artificial Intelligence 138 (1-2):1 (2002).

[7] M. R. Genesereth, E. Kao: The Herbrand Manifesto - Thinking Inside the Box, in Bassiliades N., Gottlob G., Sadri F., Paschke A., Roman D. (eds) Rule Technologies: Foundations, Tools, and Applications. RuleML 2015. Lecture Notes in Computer Science, vol 9202. Springer Cham, 2015.

[8] M. Genesereth, V. Chaudhri: Logic Programming. Synthesis Lectures on Artificial Intelligence and Machine Learning, February 2020. https://doi.org/10.2200/S00966ED1V01Y201911AIM044

[9] M. Genesereth: "Insurance Portfolio Management", Complaw Corner, Codex: The Stanford Center for Legal Informatics, 2022. https://law.stanford.edu/2022/07/30/insurance-portfolio-management/

[10] P. Hayes: Computation and deduction. Proceedings Second Symposium on Mathematical Foundations of Computer Science, Czechoslovakian Academy of Sciences, Czechoslovakia, 1973, pp. 105-118.

[11] T. Hinrichs: Extensional Reasoning. Ph.D. thesis, Computer Science Department, Stanford University, 2007.

[12] A.C. Kakas, R. Kowalski, F. Toni, The role of abduction in logic programming, in: D.M. Gabbay, C.J. Hogger, J.A. Robinson (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 5, Oxford University Press, Oxford, 1998, pp. 235-324.

[13] R. Kowalski and D. Kuehner: Linear Resolution with Selection Function, Artificial Intelligence, Vol. 2, 1971, pp. 227-60.

[14] R. Kowalski: Predicate Logic as a Programming Language. Proceedings of IFIP 1974, North Holland Publishing Company, Amsterdam, pp. 569-574, 1974.

[15] R.Kowalski: Algorithm = Logic + Control. Communications of the ACM, July 1979, Vol 22 No 7.

[16] R. Kowalski, F. Sadri: LPS - A Logic-based Production System Framework. 2009.

[17] R. Kowalski, F. Sadri: Integrating Logic Programming and Production Systems in Abductive Logic Programming Agents. 2009.

[18] Logical English as a Programming Language for the Law, ProLALA 22, 2022.

[19] Lifschitz, Vladimir (13 July 2008). "What is answer set programming?" (PDF). Proceedings of the 23rd National Conference on Artificial Intelligence. AAAI Press. 3: 1594–1597.

[20] J. McCarthy: Programs with Common Sense. Proceedings of the Teddington Conference on the Mechanization of Thought Processes, pp 75-91, London, Her Majesty's Stationary Office, 1959.

[21] J. McCarthy: Coloring Maps and the Kowalski Doctrine. V. Lifschitz (Ed.), Formalizing Common Sense: Papers by John McCarthy, 1998, pp 167-174.

[22] L. M. Pereira, A. Porto. Selective Backtracking for Logic Programs, Departamento de Informatica, Faculdade de Ciencias e Tecnologia, Universidade Nova de Lisboa, Lisboa, Portugal, 1980.

[23] K. Sagonas and T. Swift and D.S. Warren (1994), XSB as an Efficient Deductive Database Engine. Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1994.

[24] J. D. Ullman. 2000. Information integration using logical views. Theoretical Computer Science 239, 2 (2000), 189–210. https://doi.org/10.1016/S0304-3975(99) 00219-4

[25] M. van Emden, R. Kowalski: The Semantics of Predicate Logic as a Programming Language, Journal of the Association of Computing Machinery, Vol 23, No 4, October 1976, pp 733-774.

[26] D. S. Warren: Programming in Tabled Prolog. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.4635