Prolog and Answer Set Programming: Languages in Logic Programming

Sílvia Casacuberta Puig

May 12, 2020

1 Introduction

This semester, in the course *CS 152: Programming Languages* we have studied the properties and applications of different programming languages, mostly functional languages. We also devoted one lecture to logic programming, which is another type of a declarative programming paradigm based on formal logic. The main logic programming languages are Prolog and Answer Set Programming (ASP). In this essay, we will delve into these two languages and explore the features, procedures, and logical formalism behind them.

2 Prolog

2.1 The Basics

Prolog is a declarative first-order programming language, which has a wide-range of applications, from natural language processing to artificial intelligence. First-order logic is an extension of propositional logic which contains predicates, quantifiers, and variables [1, 2]. Prolog (an abbreviation of *programmation en logique* in French) was created around 1972 by Alain Colmerauer and Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses [3].

More concretely, a Prolog program consists of one or more *predicates*, and each predicate consists of one or more *clauses* [4]. If a clause has an empty body, then it is unconditionally true. Once the program is defined, we can then make queries to see if a proposition is true or not. For example, if we write:

```
cs_course(cs152).
seas_course(X) :- cs_course(X).
```

which reads as *cs152 is a cs_course* and *if X is a cs_course then X is a seas_course*, we can query

```
    seas_course(cs152).
```

and Prolog will return `true`. Some of Prolog's data types include atoms, numbers and variables, and it also has built-in lists and strings, among other data structures [5].

2.2 Pure Prolog and Extended Constructs

Pure Prolog only uses a subset of first-order logic, Horn clauses, which is Turing-complete. A clause is a *Horn clause* if it contains at most one positive literal, and a *definite clause* is a Horn clause that has exactly one positive literal [6]. Thus, pure Prolog programs consist entirely of definite clauses and there is no negation. But pure Prolog was soon extended to contain constructs such as the Cut operator and negation as failure.

2.2.1 Backtracking and the Cut Operator

How does Prolog solve the queries? It automatically chooses the facts and rules required to solve a query. To make this choice, it first tries to solve each goal in a query, going from left to right, and for each goal it attempts to match a corresponding fact or the head of a corresponding rule, called *unification* [7]. If unification fails and so no matching is possible, Prolog then *backtracks* to the last point in which a matching choice was made. Note that it might be the case that a Prolog program does *not* terminate. It is also important to note that Prolog is a *set* of clauses, not a *list*, and so order matters [8].

Motivated by this backtracking procedure, Prolog allows to define a Cut operator !, which is a goal that always succeeds but cannot be backtracked past. This ensures avoiding unwanted backtracking, so that some solutions are discarded and space is saved [9]. A cut which only improves efficiency is known as *green cut*. On the other hand, a cut that is not green is called a *red cut* [9].

2.2.2 Negation as Failure

One of the important logical features of Prolog is the so-called *negation as failure*. This means that Prolog follows the assumption that P is false if one has failed to prove that P is true [10]. Consider the following example [11]:

```
    bacherlor(X) :- male(X), not(married(X)).
    male(henry).
    male(tom).
    married(tom).
```

If we query `not(married(Who))`, Prolog will return `false` because `Who=tom`, `married(Who)` succeeds.

More formally, negation by failure shows how Prolog semantics are extended from SLD resolutions to SLDNF. Let us define these two terms: SLD is a theorem proving procedure that is complete for Horn clauses [32], and so it is the basic inference rule used in logic programming and by pure Prolog. SLDNF is then obtained by adding negation as failure [12]. Essentially, this logic framework equates "unknown" with "false".

2.2.3 Strong Negation

Strong negation is another kind of negation in the context of logic programming. John McCarthy gave a famous short example to help distinguish the two negations [13]: say that we want to express the idea "we can cross the tracks if no train is approaching". With negation as failure, this would be written as:

```
cross :- not train.
```

However, this is saying that we can cross in the absence of information about whether a train is coming or not, which is not quite capturing the same concept. With strong negation, we would write:

```
cross :- -train.
```

For notation purposes, $-$ refers to the classical negation (or strong negation), and $\sim$ refers to negation as failure.

We can also combine both types of negation, which allows us to express the *closed-world assumption* (CWA) [13]: a formal system of logic used for knowledge representation (KR) which presupposes that a statement that is true is also known to be true [19]. So, conversely, this means that a statement that is not currently known is considered to be false.

Moreover, all of these types of negation are *non-monotonic* (by failure, strong, and CWA), meaning that if we add new assumptions to a theory, we *might* invalidate some conclusion whose truth we already determined. So both Prolog and ASP are non-monotonic logic programming languages.

2.3 Other Non-Monotonic Logic Programming

Other extensions of non-monotonic logic programming have been developed, such as:

1. Abductive logic programming: this allows for some predicates, called *abducible*, to be undefined. Instead of failing in a proof when a selected subgoal fails to unify with the head of any rule, the subhead can be viewed as a hypothesis. This is similar to viewing abducibles as "askable" conditions which are treated as qualifications to answers to queries [23].

2. Concurrent logic programming: this allows for evaluating goals *in parallel* (hence the name concurrently). This allows for logic programs to take advantage of current parallel and multi-distributed systems [1]. A famous concurrent logic program is Parlog [23].

3. Constraint logic programming: this is the merger of two declarative paradigms, constraint solving and logic programming. Solving a problem with constraints means finding a way to assign values to all its variables such that all constraints are satisfied. Since constraints can be seen as relations or predicates and constraint solving can be seen as a general form of unification, it makes sense to embed constraint solving into logical programming [24]. The first CLP language was Prolog II [25].

4. Inductive logic programming: investigates the inductive construction of first-order clausal theories from examples and background knowledge [26]. Given a set of labeled examples E and a background knowledge B, an ILP system will try to find hypothesis function H that minimizes a specified loss [27]. Progol is a famous implementation of inductive logic programming (ILP).

Other extensions of logic programming are possible, for example λProlog as part of the Higher-order logic programming paradigm.

3 Answer Set Programming

3.1 Stable Model Semantics

The above discussion on negation by failure provides a good motivation for introducing stable model semantics. Consider the following rules [20]:

```
p(1).
p(2).

q(3) :- ~r(s).
r(X) :- p(X), ~q(X).
```

The above program yields two possible *answer sets*, written:

```
{p(1), p(2), p(3), q(3), r(1), r(2)}
```

and

```
{p(1), p(2), p(3), r(3), r(1), r(2)}.
```

An answer set solver is a program that takes a logic program as an input and outputs all the answers sets of that program [20]. This is exactly what ASP does.

Let us define exactly what a stable model is, i.e. when a model of a propositional formula is *stable*. Informally, a set X of atoms is a stable model of a logic program P if [30]:

1. X is a (classical) model of P, and

2. All atoms in X are justified by some rule in P.

The formal definition of stable model uses two conventions: first, a truth assignment is identified with the set of atoms that are true (as we did in the example above). Second, we identify any set X of atoms with the truth assignment that makes all elements of X true and makes all other atoms false [13].

Once we have fixed the conventions, let us see how to compute the answer set of a logic program. To do so, we need to define the term *reduct* [13]:

Definition 1. The *reduct* F^X of a propositional formula F relative to a set X of atoms is the formula obtained form F by replacing each maximal subformula that is *not* satisfied by X with $\perp$ (i.e. unconditionally false).

Then, we can formally define a *stable model*:

Definition 2. *X is a stable model of F if X is minimal among the sets satisfying F^X.*

Note that the key in these procedures is the elimination of negation; indeed, the reduct we found above does not contain any negated atom, as required.

To see an example, recall the program given at the beginning of this section, and let `S = p(1), p(2), p(3)` be the set that we want to test whether it is an answer set of the program. Then, the reduct of the program with respect to S is [20]:

```
p(1).
p(2).
q(3).
r(1) :- p(1).
r(2) :- p(2).
r(3) :- p(3).
```

In practice, to find the reduct we should first find the *grounded* instantiation of the program, see [20]. As a side note, this grounded instantiation is tightly related to the concept of *Herbrand base* and *Herbrand interpretation* in first-order logic, since the Herbrand base of a program is precisely the set ground atoms (terms without variables) built from the predicates and the Herbrand universe of the program [21].

Observe that a program with negations can have zero, one, or many stable models. For example,

```
p :- not p.
```

has no stable models. On the other hand, a program without negations, then there is only one stable model, which is minimal. Otherwise, we need to find the reduct, as seen above.

The different number of stable models lies precisely at the core difference between Prolog and ASP. In Prolog, the presence of programs with negation that do *not* have a unique stable model cause trouble and the SLDNF resolution does not terminate on them [17]. However, ASP embraces the disparity of stable models and treats the stable models of the programs as solutions to a given search program.

3.2 ASP: The Basics and Functioning

Answer set programming is a form of declarative programming which is oriented towards difficult search problems, normally NP-hard. It is based on the stable model semantics of logic programming that we discussed in the previous section [13]. This new computing paradigm that uses answer set solvers for search was introduced by Marek and Truszczyńsk in 1999.

On a syntactic level, ASP programs look like programs in Prolog, but the computational mechanisms for ASP is different: they are based on the satisfiability solvers for propositional logic [14]. An important syntactic difference between ASP and Prolog is that ASP has *choice rules*, which allows to use disjunction in the head of a rule [22]. For example, writing `p.` declares that `p` may be true or not, so this produces two answer sets: $\emptyset$ and `p`. We can also add *constrained* choice rules, for example, writing `1 {p, q, r} 2.` means that the stable model must be choosing at least one of p, q, r, but no more than 2. ASP uses the CWA approach: the given information is treated as complete information and the problem is solved under this assumption [29].

In ASP, as we started explaining in the previous section, search problems are reduced to computing stable models, and to perform search it uses answer set solvers-programs for generating stable models [13]. This search algorithm that is used to find the answer set solvers are versions of the DPLL algorithm (Davis-Putnam-Logemann-Loveland), a backtracking-based search algorithm for logic formulas, which is also used for SAT solvers [15].

Therefore, to solve a search problem with ASP we need to model the problem as a logical program. The ASP solving process then consists of two steps [18]:

1. A *grounder* converts a first-order program into a propositional logic, by systematically replacing variables with concrete values from some domain.

2. A *solver* takes the ground program and assigns truth values to atoms to obtain the stable models of the program.

Most modern answer set solving tools use front-end LPARSE or GRINGO for grounders and CMODELS or CLASP for answer set solvers [23]. These can be put together by languages such as CLINGO for example, which combines GRINGO and CLASP into a monolithic system.

As it is clear from the suitability of ASP to solve search problems with potentially finitely many solutions, ASP has a strong relation to classic NP-problems. With its declarative format, it is quite straight-forward to code all the well-known NP-hard problems: 3-Coloring, Hamiltonian Cycle, Travelling Salesman, Knapsack problem, and so on. Of course, this indicates that testing whether a finite ground logic program has a stable model is NP-complete [21].

4 Comparison between Prolog and ASP

As indicated above, these two logic programming languages are very similar, but not equivalent. In this section we will highlight the main differences between Prolog and ASP:

1. ASP and Prolog belong to two different paradigms: Prolog follows a Theorem-Proving-Based approach, where the solution to the program corresponds to the solution given by the derivation of a *query*. On the other hand, ASP follows a Model-Generation-Based Approach, where the solution is given by the *model* of the representation [18].

2. ASP is extended with choice rules, which Prolog does not have. This implies that ASP is non-deterministic and there is a possibility of making guesses [17].

3. ASP is inference-based on SAT solvers, rather than Prolog's backwards chaining [16].

4. In ASP the order of the program rules does not matter and termination is not an issue, as opposed to Prolog [17]. That is, the enhanced DLPP method used by ASP always terminates, whereas a Prolog query might enter into an infinite loop. This means that ASP is a *pure* declarative programming language, whereas Prolog is not. The Cut operator is another construct that makes Prolog not purely declarative. Other similar languages to Prolog, such as Datalog, are purely declarative, but they restrict the language [8].

5. In ASP, the answer set solvers are found using a bottom-up approach, whereas in Prolog the backtracking is performed top-down [18].

6. ASP is a *representation* language, meaning that problem specification and problem solving are decoupled. This is not the case for Prolog, which is a *programming* language because the user is allowed to exercise control [18].

Despite these differences, researchers have tried to put these two paradigms together. For example, in [23] they identify an issue of the ASP functioning called *grounding bottleneck*. This identifies a problem in the intermediate step between grounding and solving: the grounding bottleneck refers to situations where grounding results in programs are too large for the solving tools to handle effectively. In the paper, they propose a solution by implementing an approach for combining backtracking-based search algorithms of answer set solvers with the SLDNF resolution from Prolog, called ASP+PROLOG. As another example, in [31] they propose ASP-PROLOG, a tight and well-defined integration of Prolog and Answer Set Programming. The motivation is the identification of aspects of reasoning that cannot be correctly expressed in ASP, and so the idea of ASP-PROLOG is to have Prolog modules that can access any ASP module, read its content, and modify it.

References

[1] Simran, Max, and Charence, *Topics in AI - Logic Programming*, DoC Imperial College London, 2006, https://www.doc.ic.ac.uk/~cclw05/topics1/first.html.

[2] Hod Lipson, *Foundations of Artificial Intelligence*, Course CS4700 at Cornell University, 2011, https://www.cs.cornell.edu/courses/cs4700/2011fa/lectures/16_FirstOrderLogic.pdf.

[3] Robert Kowalski, *Predicate Logic as Programming Language*, Information Processing 74: North-Holland Publishing Company, 1974, https://www.doc.ic.ac.uk/~rak/papers/IFIP%2074.pdf.

[4] David Matuszek, *A Concise Introduction to Prolog*, 2012, https://www.cis.upenn.edu/~matuszek/Concise%20Guides/Concise%20Prolog.html.

[5] Peter Hancox, *Prolog and Logic Programming*, Course SEM242 at University of Birmingham, 1998, https://www.cs.bham.ac.uk/~pjh/prolog_course/se207.html.

[6] https://mathworld.wolfram.com/HornClause.html.

[7] Charles N. Fischer, *Introduction to the Theory and Design of Programming Languages*, Course CS528 at University of Wisconsin-Madison, 2008, http://pages.cs.wisc.edu/~fischer/cs538.s08/lectures/Lecture34.4up.pdf.

[8] Stephen Chong, *Programming Languages*, Course CS152 at Harvard University, 2020, https://www.seas.harvard.edu/courses/cs152/2020sp/lectures/lec21-logicprogramming.pdf.

[9] Bill Wilson, *On-line Dictionaries of Artificial Intelligence Concepts*, 2012, http://www.cse.unsw.edu.au/~billw/dictionaries/prolog/cut.html.

[10] Alan L. Ritter, *Intro to AI*, Course CSE3521 at Ohio State University, http://web.cse.ohio-state.edu/~stiff.4/cse3521/prolog.html.

[11] John R. Fisher, *Prolog Tutorial*, https://www.cpp.edu/~jrfisher/www/prolog_tutorial/2_5.html.

[12] Sergio Tessaris et al, *Reasoning Web: Semantic Technologies for Information Systems*, 5th International Summer School, 2009, https://link.springer.com/content/pdf/10.1007%2F978-3-642-03754-2.pdf.

[13] Vladimir Lifschitz, *What Is Answer Set Programming?*, https://www.cs.utexas.edu/~vl/papers/wiasp.pdf.

[14] Yuliya Lierler, *Handout on Answer Set Programming*, 2013, https://personal.utdallas.edu/~gupta/courses/lp/asphw.pdf.

[15] Alessandro Farinelli, *DPLL Method*, http://profs.sci.univr.it/~farinelli/courses/ar/slides/DPLL.pdf.

[16] Mark J. Nelson, *Why did Prolog lose steam?*, 2010, http://www.kmjn.org/notes/prolog_lost_steam.html.

[17] Thomas Eiter, *Answer Set Programming in a Nutshell*, GK Kolloqium *Mathematische Logik und Anwendungen*, Freiburg, 2008, http://gradlog.informatik.uni-freiburg.de/gradlog/slides_ak/eiter_asp.pdf.

[18] Paul Vicol, *An Introduction to Answer Set Programming*, 2015, https://www.paulvicol.com/pdfs/ASP-Lecture.pdf.

[19] Alan Smaill, *Logic Programming*, slides for a course at the University of Edinburgh, 2009, http://www.inf.ed.ac.uk/teaching/courses/lp/2009/slides/lp_theory7.pdf.

[20] Vinay K. Chaudhri, *Logic Programming*, http://web.stanford.edu/~vinayc/logicprogramming/html/answer_set_programming.html.

[21] Ilkka Niemelä, *Answer Set Programming*, https://pdfs.semanticscholar.org/ded4/0cd58de310f6ba2509b094f3b5bd9c6fc246.pdf.

[22] Chris Martens, *Notes on Answer Set Programming*, Course notes for CSC 791 Generative Methods for Game Design, 2017, http://www.cs.cmu.edu/~cmartens/asp-notes.pdf

[23] Marcello Balduccini, Yuliya Lierler, Peter Schüller, *Prolog and ASP Inference Under One Roof*, Logic programming and nonmonotonic reasoning. 12th international conference, LPNMR 2013, Corunna, Spain, 2013, https://www.researchgate.net/publication/244864312_Prolog_and_ASP_Inference_under_One_Roof.

[24] A. C. Kakas, R.A. Kowalski, and F. Toni, *Abductive Logic Programming*, Journal of Logic and Computation, Vol 2 No. 6, pp 719-770, 1993, https://www.doc.ic.ac.uk/~rak/papers/abdsurv.pdf.

[25] Marco Gavanelli1 and Francesca Rossi, *Constraint Logic Programming*, 25 Years of Logic Programming, LNCS 6125, pp. 64–86, 2010, https://www.math.unipd.it/~frossi/gulp.pdf.

[26] Stephen Muggleton and Luc de Raedt, *Inductive Logic Programming: Theory and methods*, The Journal of Logic Programming, Vol 19-20 No. 1, pp. 629-670, 1994, https://www.sciencedirect.com/science/article/pii/0743106694900353.

[27] Manoel V. M. França, *Introduction to Inductive Logic Programming*, Machine Learning Group Meeting at City University London, 2012, `https://cpb-eu-w2.wpmucdn.com/blogs.city.ac.uk/dist/8/1140/files/2014/11/ilpTalk-1xtj3xm.pdf`.

[28] Andrew Cheese, *Parallel Execution of Prolog*, Part of the Lecture Notes in Computer Science book series (LNCS, volume 586), pp. 27-47, 2005, `https://link.springer.com/chapter/10.1007%2FBFb0022708`.

[29] Tomi Janhunen and Ilkka Niemelä, *The Answer Set Programming Paradigm*, Association for the Advancement of Artificial Intelligence, 2016, `https://aaltodoc.aalto.fi/handle/123456789/35197`.

[30] Sebastian Rudolph, *Answer Set Programming: Basics*, Slides based on a lecture by Martin Gebser and Torsten Schaub, `https://iccl.inf.tu-dresden.de/w/images/1/1a/FLP-ASP-L1.pdf`.

[31] Omar Elkhatib, Enrico Pontelli, and Tran Cao Son, *ASP-PROLOG: A System for Reasoning about Answer Set Programs in Prolog*, `https://www.cs.nmsu.edu/~tson/papers/padl04.pdf`.

[32] Stefan Brass, *Pure Prolog*, Slides for the course Deductive Databases and Logic Programming at University Halle-Wittenberg, `http://users.informatik.uni-halle.de/~brass/lp06/c3_purep.pdf`.