The Logic Programming Paradigm and Prolog

Krzysztof R. Apt

Abstract

The Prolog programs presented so far are *declarative* since they admit a dual reading as a formula. The treatment of arithmetic in Prolog compromises to some extent its declarative underpinnings. However, it is difficult to come up with a better solution than the one offered by the original designers of the language. The shortcomings of Prolog's treatment of arithmetic are overcome in the constraint logic programming languages.

15.6.1 Arithmetic Operators

Prolog provides integers and floating point numbers as built-in data structures, with the typical operations on them. These operations include the usual arithmetic operators such as +, -, * (multiplication), and // (integer division).

Now, according to the usual notational convention of logic programming and Prolog, the relation and function symbols are written in the *prefix form*, that is in front of the arguments. In contrast, in accordance with their usage in arithmetic, the binary arithmetic operators are written in *infix form*, that is between the arguments. Moreover, negation of a natural number can be written in the *bracketless prefix form*, that is, without brackets surrounding its argument.

This discrepancy in the syntax is resolved by considering the arithmetic operators as built-in function symbols written in the infix or bracketless prefix form with information about their associativity and binding power that allows us to disambiguate the arithmetic expressions.

Actually, Prolog provides a means to declare an *arbitrary* function symbol as an infix binary symbol or as a bracketless prefix unary symbol, with a fixed *priority* that determines its binding power and a certain *mnemonics* that implies some (or no) form of associativity. An example of such a declaration was the line `:- op(1100, yfx, arrow).` used in the above-type assignment program. Function symbols declared in this way are called *operators*. Arithmetic operators can be thus viewed as operators predeclared in the language "prelude."

In addition to the arithmetic operators we also have at our disposal infinitely many integer constants and infinitely many floating point numbers. In what follows, by a *number*, we mean either an integer constant or a floating point number. The arithmetic operators and the set of all numbers uniquely determine a set of terms. We call terms defined in this language *arithmetic expressions* and introduce the abbreviation *gae* for ground (i.e., variable free) arithmetic expressions.

15.6.2 Arithmetic Comparison Relations

With each gae, we can uniquely associate its *value*, computed in the expected way. Prolog allows us to compare the values of gaes by means of the customary six *arithmetic comparison relations*

15.7.1 Cut

To deal with such problems, Prolog provides a low level built-in nullary relation symbol called *cut* and denoted by "!". To explain its meaning we rewrite first the above clauses using cut:

In the resulting analysis, two possibilities arise, akin to the above case distinction. First, if `B` is true (i.e., succeeds), then the cut is encountered. Its execution

- discards all alternative ways of computing `B`,
- discards the second clause, `p(x) :- T.`, as a backtrackable alternative to the current selection of the first clause.

Both items have an effect that in the current computation some clauses are not available anymore.

Second, if `B` is false (i.e., fails), then backtracking takes place and the second clause is tried. The computation proceeds now by directly evaluating `T`.

So using the cut and the above rewriting we achieved the intended effect and modelled the **if** `B` **then** `S` **else** `T` **fi** construct in the desired way.

The above explanation of the effect of cut is a good starting point to provide its definition in full generality.

Consider the following definition of a relation `p`:

Here, the i^{th} clause contains a cut atom. Now, suppose that during the execution of a query, a call `p(t)` is encountered and eventually the i^{th} clause is used and the indicated occurrence of the cut is executed. Then the indicated occurrence of ! succeeds immediately, but additionally

1. all alternative ways of computing **B** are discarded, and
2. all computations of `p(t)` using the i^{th} to k^{th} clause for `p` are discarded as backtrackable alternatives to the current selection of the i-clause.

The cut was introduced to improve the implicit control present through the combination of backtracking and the textual ordering of the clauses. Because of the use of patterns in the clause heads, the potential source of inefficiency can be sometimes hidden somewhat deeper in the program text. Reconsider for example the `QUICKSORT` program of Section 15.6 and the query `?- qs([7,9,8,1,5], Ys)`. To see that the resulting computation is inefficient, note that the second clause defining the `part` relation fails when 7 is compared with 9 and subsequently the last, third, clause is tried. At this moment 7 is again compared with 9. The same redundancy occurs when 1 is compared with 5. To avoid such inefficiencies the definition of `part` can be rewritten using cut as follows:

Of course, this improvement can be also applied to the QUICKSORT_DL program.

Cut clearly compromises the declarative reading of the Prolog programs. It has been one of the most criticized features of Prolog. In fact, a proper use of cut requires a good understanding of Prolog's computation mechanism and a number of thumb rules were developed to help a Prolog programmer to use it correctly. A number of alternatives to cut were proposed. The most interesting of them, called *commit*, entered various constraint and parallel logic programming languages but is not present in standard Prolog.

15.7.2 Ambivalent Syntax and Meta-variables

Before we proceed, let us review first the basics of Prolog syntax mentioned so far.

1. Variables are denoted by strings starting with an upper case letter or "_" (underscore). In particular, Prolog allows so-called anonymous variables, written as "_" (underscore).
2. Relation symbols (procedure identifiers), function symbols, and nonnumeric constants are denoted by strings starting with a lower case letter.
3. Binary and unary function symbols can be declared as infix or bracketless prefix operators.

Now, in contrast to first-order logic, in Prolog the *same* name can be used both for function symbols and for relation symbols. Moreover, the same name can be used for function or relation symbols of different arity. This facility is called *ambivalent syntax*. A function or a relation symbol f of arity n is then referred to as f/n. Thus, in a Prolog program, we can use both a relation symbol p/2 and function symbols p/1 and p/2 and build syntactically legal terms or atoms like p(p(a,b),c,p(X)).

In presence of the ambivalent syntax, the distinction between function symbols and relation symbols and between terms and atoms disappears, but in the context of queries and clauses, it is clear which symbol refers to which syntactic category.

The ambivalent syntax together with Prolog's facility to declare binary function symbols (and thus also binary relation symbols) as infix operators allows us to pass queries, clauses and programs as arguments. In fact, ":-/2" is declared internally as an infix operator and so is the comma ",/2" between the atoms, so each clause is actually a term. This facilitates *meta-programming*, that is, writing programs that use other programs as data.

In what follows, we shall explain how meta-programming can be realized in Prolog. To this end, we need to introduce one more syntactic feature. Prolog permits the use of variables in the positions of atoms, both in the queries and in the clause bodies. Such a variable is called then a *meta-variable*. Computation in the presence of the meta-variables is defined as before since the mgus employed can also bind the meta-variables. Thus, for example, given the legal, albeit unusual, Prolog program (that uses the ambivalent syntax facility)

the execution of the Prolog query `p(X), X.` first leads to the query `a.` and then succeeds. Here, `a` is both a constant and a nullary relation symbol.

Prolog requires that the meta-variables are properly instantiated before they are executed. In other words, they need to evaluate to a nonnumeric term at the moment they are encountered in an execution. Otherwise, a run-time error arises. For example, for the above program and the query `p(X), X, Y.`, the Prolog computation ends up in error once the query `Y.` is encountered.

15.7.3 Control Facilities

Let us now see how the ambivalent syntax in conjunction with meta-variables supports meta-programming. In this section we limit ourselves to (meta-)programs that show how to introduce new control facilities. We discuss here three examples, each introducing a control facility actually available in Prolog as a built-in. More meta-programs will be presented in the next section once we introduce other features of Prolog.

Disjunction

To start with, we can define *disjunction* by means of the following simple program:

A typical query is then `or(Q,R)`, where `Q` and `R` are "conventional queries." Disjunction is a Prolog's built-in declared as an infix operator ";/2" and defined by means of the above two rules, with "or" replaced by ";". So instead of `or(Q,R)` one writes `Q ; R`.

If-then-else

The other two examples involve the cut operator. The already discussed **if** `B` **then** `S` **else** `T` **fi** construct can be introduced by means of the now-familiar program

---

In Prolog, `if_then_else` is a built-in defined internally by the above two rules. `if_then_else(B, S, T)` is written as `B -> S;T`, where "$\rightarrow$ /2" is a built-in declared as an infix operator. As an example of its use, let us rewrite yet again the definition of the `part` relation used in the `QUICKSORT` program, this time using Prolog's `B -> S;T`. To enforce the correct parsing, we need to enclose the `B -> S;T` statement in brackets:

Note that here we had to dispense with the use of patterns in the "output" positions of `part` and reintroduce the explicit use of unification in the procedure body. By introducing yet another `B -> S;T` statement to deal with the case analysis in the second argument, we obtain a definition of the `part` relation that very much resembles a functional program:

In fact, in this program all uses of unification boil down to matching and its use does not involve backtracking. This example explains how the use of patterns often hides an implicit case analysis. By making this case analysis explicit using the **if-then-else** construct we end up with longer programs. In the end the original solution with the cut seems to be closer to the spirit of the language.

Negation

Finally, consider the negation operation not that is supposed to reverse failure with success. That is, the intention is that the query `not Q.` succeeds iff the query `Q.` fails. This operation can be easily implemented by means of meta-variables and cut as follows:

---

fail/0 is Prolog's built-in with the empty definition. Thus, the call of the parameterless procedure fail always fails.

This cryptic two-line program employs several discussed features of Prolog. In the first line, `X` is used as a meta-variable. Now consider the call `not(Q)`, where `Q` is a query. If `Q` succeeds, then the cut is performed. This has the effect that all alternative ways of computing `Q` are discarded and also the second clause is discarded. Next, the built-in fail is executed and a failure arises. Because the only alternative clause was just discarded, the query `not(Q)` fails. If, on the other hand, the query `Q` fails, then backtracking takes place and the second clause, `not(_)`, is selected. It immediately succeeds and so the initial query `not(Q)` succeeds. So this definition of not achieves the desired effect.

not/1 is defined internally by the above two line definition augmented with the appropriate declaration of it as a bracketless prefix unary symbol.

Call

Finally, let us mention that Prolog also provides an indirect way of using meta-variables by means of a built-in relation call/1. call/1 is defined internally by this rule:

---

call/1 is often used to "mask" the explicit use of meta-variables, but the outcome is the same.

15.7.4 Negation as Failure

The distinction between successful and failing computations is one of the unique features of logic programming and Prolog. In fact, no counterpart of failing computations exists in other programming paradigms.

The most natural way of using failing computations is by employing the negation operator not that allows us to turn failure into success, by virtue of the fact that the query `not Q.` succeeds iff the query `Q.` fails. This way we can use not to represent negation of a Boolean expression. In fact, we already referred informally to this use of negation at the beginning of Section 15.7.

This suggests a declarative interpretation of the not operator as a classical negation. This interpretation is correct only if the negated query always terminates and is ground. Note, in particular, that given the procedure `p` defined by the single rule `p :- p.` the query `not p.` does not terminate. Also, for the query `not(X = 1).`, we get the following counterintuitive outcome:

Thus, to generate all elements of a list Ls that differ from 1, the correct query is member(X, Ls), not(X = 1). and not not(X = 1), member(X, Ls). One usually refers to the way negation is used in Prolog as "negation as failure." When properly used, it is a powerful feature as testified by the following jewel program. We consider the problem of determining a winner in a two-person finite game. Suppose that the moves in the game are represented by a relation move. The game is assumed to be finite, so we postulate that given a position pos the query move(pos, Y). generates finitely many answers, which are all possible moves from pos. A player loses if he is in a position pos from which no move exists, i.e., if the query move(pos, Y). fails.

A position is a winning one when a move exists that leads to a losing, i.e., non-winning position. Using the negation operator, this can be written as

```
% win(X) :- X is a winning position in the two-person finite game
%       represented by the relation move.
win(X) :- move(X, Y), not win(Y).
```

This remarkably concise program has a simple declarative interpretion. In contrast, the procedural interpretation is quite complex: the query win(pos). determines whether pos is a winning position by performing a minimax search on the 0–1 game tree represented by the relation move. In this recursive procedure, the base case appears when the call to move fails. Then the corresponding call of win also fails.

15.7.5 Higher-Order Programming and Meta-Programming in Prolog

Thanks to the ambivalent syntax and meta-variables, higher-order programming and another form of meta-programming can be easily realized in Prolog. To explain this, we need two more built-ins. Each of them belongs to a different category.

Term Inspection Facilities

Prolog offers a number of built-in relations that allow us to inspect, compare, and decompose terms. One of them is =../2 (pronounced *univ*) that allows us to switch between a term and its representation as a list. Instead of precisely describing its meaning, we just illustrate one of its uses by means the following query:

The left-hand side, here Atom, is unified with the term (or, equivalently, the atom), here square([1,2,3,4], Ys), represented by a list on the right-hand side, here [square, [1,2,3,4], Ys]. In this list representation of a term, the head of the list is the leading function symbol and the tail is the list of the arguments.

Using *univ*, one can construct terms and pass them as arguments. More interestingly, one can construct atoms and execute them using the meta-variable facility. This way it is possible to realize higher-order programming in Prolog in the sense that relations can be passed as arguments. To illustrate this point, consider the following program MAP:

In the last rule, *univ* is used to construct an atom. Note the use of the meta-variable Atom. MAP is Prolog's counterpart of the familiar higher-order functional program and it behaves in the expected way. For example, given the program % square(X, Y) :- Y is the square of X. square(X, Y) :- Y is X*X. we get

---

Program manipulation facilities

Another class of Prolog built-ins makes it possible to access and modify the program during its execution. We consider here a single built-in in this category, clause/2 , that allows us to access the definitions of the relations present in the considered program. Again, consider first an example of its use in which we refer to the program MEMBER of Subsection 15.5.1.

In general, the call clause(head, body) leads to a unification of the term head :- body with the successive clauses forming the definition of the relation in question. This relation, here member, is the leading symbol of the first argument of clause/2 that has to be a non-variable.

This built-in assumes that true is the body of a fact, here member(X, [X | _]). true/0 is Prolog's built-in that succeeds immediately. Thus, its definition consists just of the fact true. This explains the first answer. The second answer is the result of unifying the term member(X,Y) :- Z with (a renaming of) the second clause defining member, namely member(X, [_ | Xs]):- member(X, Xs).

Using clause/2, we can construct Prolog interpreters written in Prolog, that is, *meta-interpreters*. Here is the simplest one:

Recall that (A,B) is a legal Prolog term (with no leading function symbol). To understand this program, one needs to know that the comma between the atoms is declared internally as a right associative infix operator, so the query A,B,C,D actually stands for the term (A,(B,(C,D))), etc.

The first clause states that the built-in true succeeds immediately. The second clause states that a query of the form A, **B** can be solved if A can be solved and **B** can be solved. Finally, the last clause states that an atomic query A can be solved if there exists a clause of the form A :- **B** such that the query **B** can be solved. The cuts are used here to enforce the a "definition by cases": either the argument of solve is true or a nonatomic query or else an atomic one.

To illustrate the behavior of the above meta-interpreter, assume that MEMBER is a part of the considered program. We then have

---

This meta-program forms a basis for building various types of interpreters for larger fragments of Prolog or for its extensions.

15.8 ASSESSMENT OF PROLOG

Prolog, because of its origin in automated theorem proving, is an unusual programming language. It leads to a different style of programming and to a different view of programming. A number of elegant Prolog programs presented here speak for themselves. We also noted that the same Prolog program can often be used for different purposes – for computing, testing or completing a solution, or for computing all solutions. Such programs cannot be easily written in other programming paradigms. Logical variables are a unique and, as we saw, very useful feature of logic programming. Additionally, pure Prolog programs have a dual interpretation as logical formulas. In this sense, Prolog supports declarative programming.

Both through the development of a programming methodology and ingenious implementations, great care was taken to overcome the possible sources of inefficiency. On the programming level, we already discussed cut and the difference lists. Programs such as `FACTORIAL` of Subsection 15.6.3 can be optimized by means of tail recursion. On the implementation level, efficiency is improved by such techniques as the last call optimization that can be used to optimize tail recursion, indexing that deals with the presence of multiple clauses, and a default omission of the occur-check (the test "x does not occur in t" in clause (5) of the Martelli–Montanari algorithm) that speeds up the unification process (although on rare occasions makes it unsound).

Prolog's only data type, the terms, is implicitly present in many areas of computer science. In fact, whenever the objects of interest are defined by means of grammars, for example first-order formulas, digital circuits, programs in any programming language, or sentences in some formal language, these objects can be naturally defined as terms. Prolog programs can then be developed starting with this representation of the objects as terms. Prolog's support for handling terms by means of unification and various term inspection facilities becomes handy. In short, symbolic data can be naturally handled in Prolog.

The automatic backtracking becomes very useful when dealing with search. Search is of paramount importance in many artificial intelligence applications and backtracking itself is most natural when dealing with the NP-complete problems. Moreover, the principle of "computation as deduction" underlying Prolog's computation process facilitates formalization of various forms of reasoning in Prolog. In particular, Prolog's negation operator `not` can be naturally used to support nonmonotonic reasoning. All this explains why Prolog is a natural language for programming artificial intelligence applications, such as automated theorem provers, expert systems, and machine learning programs where reasoning needs to be combined with computing, game playing programs, and various decision support systems.

Prolog is also an attractive language for computational linguistics applications and for compiler writing. In fact, Prolog provides support for so-called definite clause grammars (DCG). Thanks to this, a grammar written in the DCG form is already a Prolog program that forms a parser for this grammar. The fact that Prolog allows one to write executable specifications makes it also a useful language for rapid prototyping, in particular in the area of meta-programming.

For the sake of a balanced presentation let us discuss now Prolog's shortcomings.

Lack of Types

Types are used in programming languages to structure the data manipulated by the program and to ensure its correct use. In Prolog, one can define various types like binary trees and records. Moreover, the language provides a notation for lists and offers a limited support for the type of all numbers by means of the arithmetic operators and arithmetic comparison relations. However, Prolog does not support types in the sense that it does not check whether the queries use the program in the intended way.

Because of this absence of type checking, type errors are easy to make but difficult to find. For example, even though the `APPEND` program was meant to be used to concatenate two lists, it can also be used with nonlists as arguments:

---

and no error is reported. In fact, almost every Prolog program can be misused. Moreover, because of lack of type checking some improvements of the efficiency of the implementation cannot be carried out and various run-time errors cannot be prevented.

Subtle Arithmetic

We discussed already the subtleties arising in presence of arithmetic in Section 15.6. We noted that Prolog's facilities for arithmetic easily lead to run-time errors. It would be desirable to discover such errors at compile time but this is highly nontrivial.

Idiosyncratic Control

Prolog's control mechanisms are difficult to master by programmers accustomed to the imperative programming style. One of the reasons is that both bounded iteration (the `for` statement) and unbounded iteration (the `while` statement) need to be implemented by means of the recursion. For example, a nested `for` statement is implemented by means of nested tail recursion that is less easy to understand. Of course, one can introduce both constructs by means of meta-programming, but then their proper use is not enforced because of the lack of types. Additionally, as already mentioned, cut is a low-level mechanism that is not easy to understand.

Complex Semantics of Various Built-ins

Prolog offers a large number of built-ins. In fact, the ISO Prolog Standard describes 102 built-ins. Several of them are quite subtle. For example, the query `not(not Q).` tests whether the query `Q.` succeeds and this test is carried out without changing the state, i.e., without binding any of the variables. Moreover, it is not easy to describe precisely the meaning of some of the built-ins. For example, in the ISO Prolog Standard the operational interpretation of the **if-then-else** construct consists of 17 steps.

No Modules and No Objects

Finally, even though modules exist in many widely used Prolog versions, neither modules nor objects are present in ISO Prolog Standard.as This makes it difficult to properly structure Prolog programs and reuse them as components of other Prolog programs. It should be noted that thanks to Prolog's support for meta-programming, the object-programming style can be mimicked in Prolog in a simple way. But no compile-time checking of its proper use is then enforced then and errors in the program design will be discovered at best at the run-time. The same critique applies to Prolog's approach to higher-order programming and to meta-programming.

Of course, these limitations of Prolog were recognized by many researchers who came up with various good proposals on how to improve Prolog's control, how to add to it (or how to infer) types, and how to provide modules and objects. Research in the field of logic programming also has dealt with the precise relation between the procedural and declarative interpretation of logic programs and a declarative account of various aspects of Prolog, including negation and meta-programming. Also verification of Prolog programs and its semantics were extensively studied.

However, no single programming language proposal emerged yet that could be seen as a natural successor to Prolog in which the above shortcomings are properly solved. The language that comes closest to this ideal is Mercury (see http://www.cs.mu.oz.au/research/mercury/). Colmerauer designed a series of successors of Prolog, Prolog II, III, and IV that incorporated various forms of constraint processing into this programming style.

When assessing Prolog, it is useful to have in mind that it is a programming language designed in the early 1970s (and standardized in the 1990s). The fact that it is still widely used and that new applications for it keep being found testifies to its originality. No other programming language succeeded to embrace first-order logic in such an effective way.

15.9 BIBLIOGRAPHIC REMARKS

For those interested in learning more about the origins of logic programming and of Prolog, the best place to start is Colmerauer and Roussel's account (The Birth of Prolog, in Bergin and Gibson, History of Programming Languages, ACM Press/Addison-Wesley, 1996, pp. 331–367). There a number of excellent books on programming in Prolog. The two deservedly most successful are Bratko (*PROLOG Programming for Artificial Intelligence*, Addison-Wesley, 2001) and Sterling and Shapiro (*The Art of Prolog*, MIT Press, 1994). The work by O'Keefe (*The Craft of Prolog*, MIT Press, 1990) discusses in depth the efficiency and pragmatics of programming in Prolog. The work by Aït-Kaci (*Warrens' Abstract Machine*, MIT Press, 1991. Out of print. Available at http://www.isg.sfu.ca/~hak/documents/wam.html) is an outstanding tutorial on the implementation of Prolog.

15.10 CHAPTER SUMMARY

We discussed the logic programming paradigm and its realization in Prolog. This paradigm has contributed a number of novel ideas in the area of programming languages. It introduced unification as a computation mechanism and it realized the

Table 15.1.

Logic Programming	Imperative Programming
equation solved by unification	assignment
relation symbol	procedure identifier
term	expression
atom	procedure call
query	program
definition of a relation	procedure declaration
local variable of a rule	local variable of a procedure
logic program	set of procedure declarations
"," between atoms	sequencing (";")
substitution	state
composition of substitutions	state update

concept of "computation as deduction". Additionally, it showed that a fragment of first-order logic can be used as a programming language and that declarative programming is an interesting alternative to structured programming in the imperative programming style.

Prolog is a rule-based language but thanks to a large number of built-ins it is a general purpose programming language. Programming in Prolog substantially differs from programming in the imperative programming style. Table 15.1 may help to relate the underlying concepts used in both programming styles.

Acknowledgements

Maarten van Emden and Jan Smaus provided K.R. Apt with useful comments on this chapter.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine.* MIT Press, 1991. Out of print. Available via `http://www.isg.sfu.ca/~hak/documents/wam.html`.

[2] I. Bratko. *PROLOG Programming for Artificial Intelligence.* International Computer Science Series. Addison-Wesley, third edition, 2001.

[3] A. Colmerauer and Ph. Roussel. The birth of Prolog. In Thomas J. Bergin and Richard G. Gibson, editors, *History of Programming Languages*, pages 331–367. ACM Press/Addison-Wesley, 1996.

[4] H.B. Curry and R. Feys. *Combinatory Logic, Volume I, Studies in Logic and the Foundation of Mathematics.* North-Holland, Amsterdam, 1958.

[5] International Standard. Information Technology — Programming Language — Prolog, Part 1: General Core, 1995. ISO/IEC DIS 13211-1:1995(E).

[6] R.A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP'74*, pages 569–574. North-Holland, 1974.

[7] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[8] R.A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[9] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[10] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Mass., second edition, 1994.