Logic Programming

K.R. Apt

1. Introduction

Logic programming (in short LP) is a simple, yet powerful formalism suitable for programming and for knowledge representation. It was introduced in 1974 by R. Kowalski. LP grew out of an earlier work on automatic theorem proving based on the resolution method. The major difference is that LP can be used not only for proving but also for computing. In fact, LP offers a new programming paradigm, which was originally realized in Prolog, a programming language introduced in early seventies by a group led by A. Colmerauer.

After an initially slow start LP grew twenty years later to an impressive field in computer science, in which by now a couple of thousand articles have been published. Recently, *The Journal of Logic Programming* celebrated its tenth year anniversary. A couple of annual conferences are nowadays taking place and interest in the subject does not seem to be waning. On the contrary. The logic programming paradigm has inspired the design of new programming languages, like CHIP and Gödel, which have been successfully used to tackle various computationally complex problems. These languages attempt to overcome a number of Prolog deficiencies, visibly awkward use of arithmetic, ad hoc control features and lack of types.

One of the reasons for this interest in LP is its simplicity combined with versatility. LP strongly relies on mathematical logic which developed its own methods and techniques and can provide a rigorous mathematical frame-

work for LP. In many cases these methods have to be fine tuned and appropriately modified to be useful in LP. It should be added here that some basic concepts of LP, like unification, were developed earlier by computer scientists working in the field of automated reasoning.

Efficient implementation of Prolog and its extensions, development of appropriate programming methodology and techniques, that aim at better understanding of the logic programming paradigm, and finally design of various successors and/or improvements of Prolog turned out to be an exciting and highly non-trivial field calling for new solutions and fresh insights.

Prolog was originally designed as a programming language for natural language processing. But it soon turned out that other natural applications for the logic programming paradigm exist. Current applications of LP involve such diverse areas as molecular biology, design of VLSI systems, representation of legislation, and option trading. These applications exploit the fact that knowledge about certain domains can be conveniently written down as facts and rules which directly translate into executable logic programs.

These three aspects of LP—theory, programming and applications—grew together and often influenced each other. This versatility of LP makes it an attractive subject to study and an interesting field to work in.

2. Declarative programming

LP allows us to write programs and compute using them. There are two natural interpretations of a logic program. The first one, called a *declarative interpretation*, is concerned with the question *what* is being computed, whereas the second one, called a *procedural interpretation*, explains *how* the computation takes place. Informally, we can say that declarative interpretation is concerned with the *meaning*, whereas procedural interpretation is concerned with the *method*.

These two interpretations are closely related to each other. The first interpretation helps us to better understand the second and is a major reason why LP is an attractive formalism for programming. The fact that when designing a logic program one can rely on its declarative interpretation explains why LP supports *declarative programming*. Loosely speaking, declarative programming can be described as follows. *Specifications*, when written in an appropriate format, can be used as a *program*. Then the desired conclusions follow *logically* from the program. To compute these conclusions some *computation mechanism* is available.

Now 'thinking' declaratively is in general much easier than 'thinking' procedurally. So declarative programs are often simpler to understand, develop and modify. In fact, in some situations the specification of a problem in the appropriate format forms already the algorithmic solution to the problem. In other words, declarative programming makes it possible to write exe-

cutable specifications. It should be added however, that in practice the programs obtained in this way are often inefficient, so this approach to programming has to be coupled with appropriate use of program transformations and various optimization techniques.

This dual interpretation of logic programs also accounts for the double use of LP—as a formalism for programming and for knowledge representation, and explains the importance of LP in the field of artificial intelligence.

3. DECLARATIVE PROGRAMMING IN PROLOG

Prolog differs from LP in several small, but important aspects. In particular, both the selection rule ('choose left first') and the search strategy (depth first search) are fixed.

To illustrate the declarative programming in Prolog we now present three examples. These were on purpose chosen short and simple.

Example 1: a sequence problem

Consider the following problem: arrange three 1's, three 2's, ..., three 9's in sequence so that for all $i \in [1, 9]$ there are exactly i numbers between successive occurrences of i. Figure 1 shows the program that solves this problem in Prolog and the output showing all 6 solutions.

We see that the Prolog solution to the problem is an almost literal formalization of its formulation.

Example 2: typing of lambda terms

Consider the typed lambda calculus and Curry's system of type assignment (see H.B. Curry and R. Feys [4]). It involves statements of the form $x : t$ which should be read as 'term x has type t'. Finite sequences of such statements often called *environments* are denoted below by E. The following three rules allow us to assign types to lambda terms:

$$\frac{x : t \in E}{E \vdash x : t}$$

$$\frac{E \vdash m : s \rightarrow t \ , \quad E \vdash n : s}{E \vdash (m\ n) : t}$$

$$\frac{E \ , \ x : s \vdash m : t}{E \vdash (\lambda x.m) : s \rightarrow t}$$

These rules directly translate into the Prolog program given in figure 2. For the sake of this program lambda terms are encoded as first-order terms. To this end the unary function symbol `var` and two binary function symbols, `lambda` and `apply` are used. The lambda term x is translated to the term

Arrange three 1's, three 2's, ..., three 9's in sequence so that for all $i \in [1, 9]$ there are exactly i numbers between successive occurrences of i.

```
% sequence(Xs) ← Xs is a list of 27 elements.
sequence([_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]).

% question(Ss) ← Ss is a list of 27 elements forming the desired sequence.
question(Ss) ←
   sequence(Ss),
   sublist([1,_,1,_,1], Ss),
   sublist([2,_,_,2,_,_,2], Ss),
   sublist([3,_,_,_,3,_,_,_,3], Ss),
   sublist([4,_,_,_,_,4,_,_,_,_,4], Ss),
   sublist([5,_,_,_,_,_,5,_,_,_,_,_,5], Ss),
   sublist([6,_,_,_,_,_,_,6,_,_,_,_,_,_,6], Ss),
   sublist([7,_,_,_,_,_,_,_,7,_,_,_,_,_,_,_,7], Ss),
   sublist([8,_,_,_,_,_,_,_,_,8,_,_,_,_,_,_,_,_,8], Ss),
   sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], Ss).

% append(Xs, Ys, Zs) ← Zs is the result of concatenating the lists Xs and Ys.
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) ← append(Xs, Ys, Zs).

% sublist(Xs, Ys) ← Xs is a sublist of the list Ys.
sublist(Xs, Ys) ← append(_, Zs, Ys), append(Xs, _, Zs).

| ?- question(Ss).

Ss = [1,9,1,2,1,8,2,4,6,2,7,9,4,5,8,6,3,4,7,5,3,9,6,8,3,5,7];

Ss = [1,8,1,9,1,5,2,6,7,2,8,5,2,9,6,4,7,5,3,8,4,6,3,9,7,4,3];

Ss = [1,9,1,6,1,8,2,5,7,2,6,9,2,5,8,4,7,6,3,5,4,9,3,8,7,4,3];

Ss = [3,4,7,8,3,9,4,5,3,6,7,4,8,5,2,9,6,2,7,5,2,8,1,6,1,9,1];

Ss = [3,4,7,9,3,6,4,8,3,5,7,4,6,9,2,5,8,2,7,6,2,5,1,9,1,8,1];

Ss = [7,5,3,8,6,9,3,5,7,4,3,6,8,5,4,9,7,2,6,4,2,8,1,2,1,9,1];
```

Figure 1. The sequence problem.

```
curry(E, var(X), T) ← member([X, T], E).
curry(E, apply(M, N), T) ← curry(E, M, S → T),
                           curry(E, N, S).
curry(E, lambda(X, M), S → T) ← curry([[X, S] | E], M, T).

member(X, [Y | Xs]) ← X ≠ Y, member(X, Xs).
member(X, [X | Xs]).
```

Figure 2. The type assignment program.

`var(x)`, the lambda term $(m\ n)$ to the term `apply(m, n)`, and the lambda term $\lambda x.m$ to the term `lambda(x, m)`. The subtle point is that according to Prolog convention, lower case letters stand for constants, so `var(x)` is a ground term (i.e. a term without variables), etc. For example, the lambda term $\lambda x.\ (x\ x)$ translates to `lambda(x, apply(var(x), var(x)))`.

Now, the program in figure 2 can be used to compute a type assignment to a lambda term, if such an assignment exists, and to report a failure if such an assignment does not exist. To this end, given a lambda term s, it suffices to use the query `curry([], t, T)`, where `t` is the translation of s to a first-order term.

The problem of computing a type assignment for lambda terms was posed and solved by Curry [4]. It is considered to be an advanced topic in the theory of lambda calculus and the foundations of functional programming. The solution in Prolog given in figure 2 is completely elementary.

Example 3: temporal reasoning

In [5] S. Hanks and D. McDermott discussed a simple problem in temporal reasoning, a branch of non-monotonic reasoning. It became known in the literature as the 'Yale Shooting Problem'. Hanks and McDermott's interest in this problem arose from the fact that apparently all theories about non-monotonic reasoning, when used to formalize this problem, led to too weak conclusions. The problem has been extensively discussed in the literature and several solutions to it have been proposed. In Hanks and McDermott [5] some of these solutions are discussed and critically evaluated.

We present here a particularly simple solution to the above problem by means of logic programming. First, let us explain the problem. Consider a single individual who in any situation can be either `alive` or `dead`, and a gun that can be either `loaded` or `unloaded`. The following statements are stipulated:

1. At some specific situation s_0 the person is alive.
2. The gun becomes loaded any time a `load` event happens.

$$
\begin{array}{l}
holds(alive, s_0), \\
\forall s\ holds(loaded, results(load, s)), \\
\forall s\ (holds(loaded, s) \rightarrow ab(alive, shoot, s) \wedge holds(dead, result(shoot, s))), \\
\forall f \forall e \forall s\ ((holds(f, s) \wedge \neg ab(f, e, s)) \rightarrow holds(f, result(e, s))).
\end{array}
$$

Figure 3. The Yale Shooting Problem: the original formulation.

3. Any time the person is shot with a loaded gun, he becomes dead. Moreover, the fact of staying alive is abnormal with respect to the event of being shot with a loaded gun.

4. Facts which are not abnormal with respect to an event remain true.

To formalize these statements Hanks and McDermott [5] used so-called *situation calculus* in which one distinguishes three entities: facts, events and situations, denoted respectively by the letters f, e, s, and the function *result* such that for an event e and a situation s the term $result(e, s)$ denotes the situation resulting from the occurrence of e in s. These four statements lead to the four formulas given in figure 3.

The problem was to find a way of interpreting these formulas so that statements like

$$holds(dead,\ result(shoot,\ result(wait,\ result(load,\ s_0))))$$

could be proved. Here *wait* is a new event whose occurrence is supposed to have no effect on the truth of the considered facts.

The solution to the Yale Shooting Problem in Prolog is completely straightforward and shown in figure 4. This program is an almost literal translation of the above formulas to Prolog syntax. (To enhance readability we use in it the list notation `[e | s]` of Prolog instead of $result(e, s)$ and denote the initial situation by the empty list `[ ]`.)

In contrast to the solutions in other formalisms, the Prolog solution can

```
holds(alive, []).
holds(loaded, [load | Xs]).
holds(dead, [shoot | Xs]) ← holds(loaded, Xs).
ab(alive, shoot, Xs) ← holds(loaded, Xs).
holds(Xf, [Xe | Xs]) ← ¬ ab(Xf, Xe, Xs), holds(Xf, Xs).
```

Figure 4. A program solving the Yale Shooting Problem.

be used not only to model the problem but also to compute answers to the relevant queries. For example, we have

```
| ?- holds(dead, [shoot, wait, load]).

yes

| ?- holds(dead, [wait, load]).

no
```

Also, using the theory of logic programming, it is possible to provide a natural semantics to this program in the form of a unique model, which admits several natural characterizations and allows us to predict the correct answers to the queries. In [6] E. Marchiori studied an extension of Prolog with so-called constructive negation. This allowed her to deal with more complex queries which Prolog does not handle correctly, like `holds(alive, [X, Y])`. By means of constructive negation it yields two answers `X` $\neq$ `shoot` and `Y` $\neq$ `load`, which is justified from the semantic point of view.

4. PROGRAM VERIFICATION AND PROLOG

The usual way of explaining that a program is correct is that it meets its specifications. This statement has a clear intention but is somewhat imprecise so we shall be more specific in a moment. Correctness of programs is important, both from the point of view of software reliability as from the point of view of software development. Program verification is the formal activity whose aim is to ensure correctness of programs. It has a history spanning a quarter of the century.

In the case of logic programming the declarative interpretation reduces the issue of program correctness to an analysis of the program from the logical point of view. In this analysis the computation mechanism can be completely disregarded. This is an important reduction which significantly simplifies the task of program verification.

In the case of Prolog it is natural to base the program verification on the theory of logic programming. Because of the differences between Prolog and logic programming this theory has to be appropriately modified and revised. Moreover, due to several 'non-declarative' features of Prolog, a declarative interpretation of Prolog programs is, to say the least, problematic. To cope with this problem we determined in our studies a large subset of Prolog and showed that for programs written in this subset it is possible to reason about their correctness by a combination of syntactic analysis and declarative interpretation.

In our approach we dealt with various program properties which are crucial for ensuring proper functioning of these programs. In particular, we

considered:

- Termination.

  This means that the program under consideration should terminate for the appropriate queries.

- Partial correctness.

  This means that the program under consideration should deliver correct answer for the appropriate queries.

- Absence of run-time errors.

  In the case of Prolog these are:

  - absence of the so-called occur-check problem (explained below),
  - absence of errors in presence of arithmetic expressions.

The resulting framework is simple to use and readily applicable to most of the well-known Prolog programs. Moreover, several aspects of the proposed methods can be automated.

4.1. Termination

As an example we explain here the approach to termination of simple Prolog programs due to K.R. Apt and D. Pedreschi [1].

We need some introductory notions. We assume here that all programs and queries are written in a fixed, 'universal', language defined by, say, a Prolog manual.

Definition

- A *level mapping* is a function $|\ |$ from ground atoms (i.e. atomic formulas with no variables) to natural numbers.

- An atom A is *bounded w.r.t.* $|\ |$, if $|\ |$ is bounded on the set of all ground instances of A.

- A clause c is *acceptable w.r.t.* $|\ |$ *and* an interpretation I, if

  - $I \models c$ (I is a model of c),
  - for all ground instances $A \leftarrow \mathbf{A}, B, \mathbf{B}$ of c such that $I \models \mathbf{A}$ $|A| > |B|$.

- A program is *acceptable w.r.t.* $|\ |$ *and* I, if every clause of it is.

Then the following result holds.

Theorem. Suppose that

- P is acceptable w.r.t. $|\,|$ and I,
- A is bounded w.r.t. $|\,|$.

Then all Prolog computations of A w.r.t. P are finite.

Let now `ls(.)` (for *listsize*) be a function from ground terms to natural numbers defined by induction as follows:

$$\texttt{ls}([x|xs]) = \texttt{ls}(xs) + 1,$$
$$\texttt{ls}(f(x_1, \ldots, x_n)) = 0 \text{ if } f \neq [\,.\,|\,.\,].$$

To see a simple use of the above theorem consider the program of figure 1. It is easily seen to be acceptable w.r.t. the level mapping defined by:

$$\begin{aligned} |\texttt{question(xs)}| &= 50, \\ |\texttt{sequence(xs)}| &= 0, \\ |\texttt{sublist(xs,ys)}| &= \texttt{ls(xs)} + \texttt{ls(ys)} + 1, \\ |\texttt{append(xs,ys,zs)}| &= \min(\texttt{ls(xs)},\ \texttt{ls(zs)}), \end{aligned}$$

and any model I of it such that for a ground term `s`

$$I \models \texttt{seq(s)} \text{ iff } \texttt{s} \text{ is a list of 27 elements.}$$

Note that `question(Ss)` is bounded w.r.t. $|\,|$, so we conclude that all Prolog computations of `question(Ss)` are finite.

A natural modification of this approach to programs that use negation can be used to deal with termination of the Prolog programs given in figures 2 and 4.

4.2. Occur-check problem

The occur-check is a special test used in the unification algorithm, a cornerstone of Prolog's computation mechanism. In most Prolog implementations it is omitted for efficiency reasons. This omission affects the unification algorithm and introduces a possibility of divergence. This is obviously an undesired situation.

In our work (see Apt and A. Pellegrini [2]) we provided easy to check syntactic conditions which can be verified mechanically and which imply that the occur-check can be safely omitted for a given program and query. For example, the programs given in figures 1 and 4 are safe from the occur-check problem for the queries used.

In contrast, the program given in figure 2 leads to problems. In particular, for the query `curry([], lambda(x, apply(var(x), var(x))), T)` the omission of the occur-check causes divergence. In our studies we showed how this problem can be taken care of by means of a simple program transformation which inserts calls of the built-in unification predicate (with the

occur-check test) into the program text. In the case of the program given in figure 2 it suffices to modify the second and the last clause as follows:

```
curry(E, apply(M, N), T) ← curry(E, M, S → T),
                           curry(E, N, Z),
                           Z =_oc S.

member(X, [Z | Xs]) ← Z =_oc X.
```

Here '$=_{oc}$' is the unification predicate with the occur-check test. The resulting program can then be used for the queries of interest. For example, in the case of the term $\lambda x.\,(x\ x)$ we use the query `curry([], lambda(x, apply(var(x), var(x))), T)`. This query finitely fails:

```
| ?-  curry([],  lambda(x, apply(var(x), var(x))), T)

no
```

which confirms that the original lambda term has no type assignment.

4.3. Delay declarations

One of the striking features of logic programs is that they can be easily parallelized. For example, by adding to the program of figure 2 the so-called delay declaration:

```
DELAY append(_, _, Z) UNTIL nonvar(Z).
```

we obtain a program with a large degree of parallelism. The above declaration defers the selection of the `append`-atoms until their last argument is not a variable. By default, no restrictions are imposed on the selection of other atoms.

So the use of delay declarations replaces the Prolog selection rule by a non-deterministic selection rule which dynamically determines which atoms can be selected. In the executions of the resulting programs dynamic networks of processes are created that communicate asynchronously by means of multiparty channels. In the case of the program of figure 1 up to nineteen processes can be created during its executions.

The delay declarations allow us to impose a synchronization on the actions of a logic program in a concise way. Programs augmented by the delay declarations can be translated in a straightforward way into other concurrent languages based on the logic programming paradigm.

In our recent publications, we showed how correctness of such parallel programs can be established by a natural modification of the methods originally developed for Prolog programs.

```
rel sequence:  array [1..27] of [1..9].
sequence(A) ← ∀I ∈ [1..9] ∃J ∈ [1..25-2I]
    (A[J] = I, A[J+I+1] = I, A[J+2I+2] = I)).
```

Figure 5. A simple solution to the problem from figure 1.

4.4. Language extensions

In our recent work [3] we studied language extensions which involve iteration and arrays. Iteration is implemented by means of bounded quantification. We noticed that the use of iteration within the logic programming paradigm often leads to substantially simpler programs which are closer to specifications and are guaranteed to terminate.

As an example consider the solution to the problem from figure 1 given in figure 5. It is very close to the problem specification and much simpler than the one given in figure 1. The range `J ∈ [1..25-2I]` comes from the requirement that the indices `J`, `J+I+1`, `J+2I+2` should lie within `[1..27]`.

REFERENCES

1. K. R. APT, D. PEDRESCHI (1993). Reasoning about termination of pure Prolog programs. *Information and Computation 106(1)*, 109–157.
2. K. R. APT, A. PELLEGRINI (1994). On the occur-check free Prolog programs. *ACM Toplas 16(3)*, 687–726.
3. K. R. APT (1995). Arrays, bounded quantification and iteration in logic and constraint logic programming. M. ALPUENTE FRASNEDO, M. I. SESSA (eds.). *1995 Joint Conference on Declarative Programming (GULP-PRODE '95)*. University of Salerno, Italy, Invited Lecture, 19–35.
4. H.B. CURRY, R. FEYS (1958). *Combinatory Logic, Volume I, Studies in Logic and the Foundation of Mathematics*, North-Holland, Amsterdam.
5. S. HANKS, D. MCDERMOTT (1987). Nonmonotonic logic and temporal projection. *Artificial Intelligence 33*, 379-412.
6. E. MARCHIORI (1995). On termination of general logic programs w.r.t. constructive negation. *The Journal of Logic Programming*. Accepted for publication.