Logic Programming

Robert Kowalski

MIT Encyclopaedia of Cognitive Science
(eds. R A Wilson and F C Keil)
MIT Press, 1999, pp. 484-486.

Logic programming is the use of logic to represent programs and of deduction to execute programs in logical form. To this end, many different forms of logic and many varieties of deduction have been investigated. The simplest form of logic, which is the basis of most logic programming, is the *Horn clause* subset of logic, where programs consist of sets of implications: A_0 if A_1 and A_2 and ... and A_n. Here each A_i is a simple (i.e. *atomic*) sentence.

It is usual to write such implications backward. This is because deduction by backward reasoning interprets them as *procedures*: to solve the goal A_0, solve the subgoals A_1 and A_2 and ... and A_n. The number of conditions, n, can be 0, in which case the implication is simply a *fact*, which behaves as a procedure which solves goals directly without introducing subgoals.

The procedural interpretation of implications can be used for *declarative programming*, where the programmer describes information in logical form and the deduction mechanism uses backward reasoning to achieve problem solving behavior. Consider, for example, the implication

> X is a citizen of the USA
> if X was born in the USA.

Backward reasoning treats the sentence as a procedure

> To show X is a citizen of the USA,
> show that X was born in the USA.

In fact, backward reasoning can be used, not only to show a particular person is a citizen, but it can also be used to find people who are citizens by finding people who were born in the USA.

Logic programs are *non-deterministic* in the sense that many different procedures might apply to the same goal. For example, naturalisation and being the child of a citizen provide alternative procedures for obtaining citizenship.

The non-deterministic exploration of alternative procedures, to find one or more which solve a given goal, can be performed by many different search strategies. In the logic programming language Prolog, search is performed depth-first, trying procedures one at a time, in the order in which they are written, backtracking in case of failure.

Declarative programming is an ideal. The programmer specifies *what* the problem is and what knowledge should be used in solving problems, and the computer determines *how* the knowledge should be used.

The declarative programming ideal works in some cases where the knowledge has a particularly simple form. But it fails in many others, where non-determinism leads to an excessively inefficient search. This failure has lead many programmers to reject logic programming in favor of conventional, imperative programming languages.

The following example shows the kind of problem that can arise with declarative programming:

There is a path from X to Y if there is a step from X to Y.

There is a path from X to Y if there is a path from X to Z
and there is a path from Z to Y.

Given no other information and the goal of showing whether there is a path from a node **a** to to a node **b**, Prolog fails to find a step from **a** to **b**, using the first procedure. It therefore tries to find a path from **a** to some Z, using the second procedure. There is no step from **a** to any Z, using the first procedure. So Prolog tries to find **a** path from **a** to some Z', using the second procedure. Continuing in this way, it goes into an infinite loop, looking for paths from **a** to Z, to Z', to Z'',....

The alternative to rejecting logic programming because of such problems or of restricting it to niche applications, is for the programmer to take responsibility for *both* the declarative correctness *and* the procedural efficiency of programs.

The following is such a correct and efficient logic program for the path-finding problem. It incrementally constructs a path of nodes already visited and ensures that no step is taken which revisits a node already in the path. The goal of showing there is a path from X to Y is reformulated as the goal of extending the path consisting of the single node X to a path ending at Y. For simplicity, a path is regarded as a trivial extension of itself.

the path P can be extended to a path ending at Y
if P ends at Y.

the path P can be extended to a path ending at Y
if P ends at X
and there is a step from X to Z
and Z is not in P
and P' extends P by adding Z to the end of P
and the path P' can be extended to a path ending at Y.

It is usual to interpret the negation in a condition, such as "Z is not in P" above, as *negation by failure*: a subgoal of the form "not A" is deemed to hold if and only if the positive subgoal "A" cannot be shown to hold. Programs containing such negative conditions, extending the Horn clause subset of logic programming, are called *normal logic programs,*

The use of negation as failure renders logic programming a NONMONOTONIC LOGIC: The addition of new information may cause a previously justified conclusion to be withdrawn. A simple example is the sentence

> X is innocent if not X is guilty.

The condition “not X is guilty” is interpreted as “it cannot be shown that X is guilty”.

Many extensions of normal logic programming have been investigated. Among the most important of these is the extension to include METAREASONING. For example, the following implication is a fragment of a metalevel logic program which can be used to reason about the interval of time for which a conclusion holds:

> “R” holds for interval I
> if “R if S” holds for interval I_1
> and “S” holds for interval I_2
> and I is the intersection of I_1 and I_2.

Similar metalevel programs are used to construct explanations and to implement resource-bounded reasoning and reasoning with uncertainty.

Among the other extensions of logic programming being investigated are extensions to incorporate constraint processing, a second “strong” form of negation, disjunctive conclusions and abductive reasoning. Methods are being developed both to execute programs efficiently and to transform inefficient programs into more efficient ones. Applications range from natural language processing and legal reasoning to commercial knowledge management systems and parts of the Windows NT operating system.

Apt, K. and Turini, F. eds. (1995) Meta-Logics and Logic Programming. MIT Press.

Bratko, I. (1988) Prolog Programming for Artificial Intelligence. Addison Wesley.

Clark, K. L. (1978) Negation by failure, in "Logic and databases", Gallaire, H. and Minker, J. [eds], Plenum Press, (293-322).

Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P. (1973) Un systeme de communication homme-machine en Francais. Research report, Groupe d’Intelligence Artificielle, Universite d’Aix-Marseille II, Luminy.

Flach, P. (1994) Simply Logical: Intelligent Reasoning by Example. John Wiley and Sons.

Gabbay, D., Hogger, C. and Robinson, J.A. (1993) Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 1: Logic Foundations. Oxford: Clarendon Press.

Gabbay, D. , Hogger, C. and Robinson, J.A. (1997) Handbook of Logic in Artificial Intelligence and Logic Programming. Vol. 5: Logic Programming. Oxford: Clarendon Press.

Gillies, D. (1996) Artificial Intelligence and Scientific Method. Oxford University Press.

Kowalski, R.(1974) "Predicate Logic as Programming Language", in *Proceedings IFIP Congress,* Stockholm, North Holland Publishing Co. (569-574).

Kowalski, R.(1979) "Logic for Problem Solving", North Holland Elsevier.

Kowalski, R.(1992) "Legislation as Logic Programs", in *Logic Programming in Action* (eds. G. Comyn, N. E. Fuchs, M. J. Ratcliffe), Springer-Verlag (203-230).

Lloyd J. W. (1987): "Foundations of logic programming", second extended edition, Springer-Verlag.